Report No. 398

A LIMITED CONNECTION ARITHMETIC UNIT

by

Michael John Pisterzi

June 1, 1970

DEPARTMENT OF COMPUTER SCIENCE
UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN · URBANA, ILLINOIS

Report No. 398

A LIMITED CONNECTION ARITHMETIC UNIT *

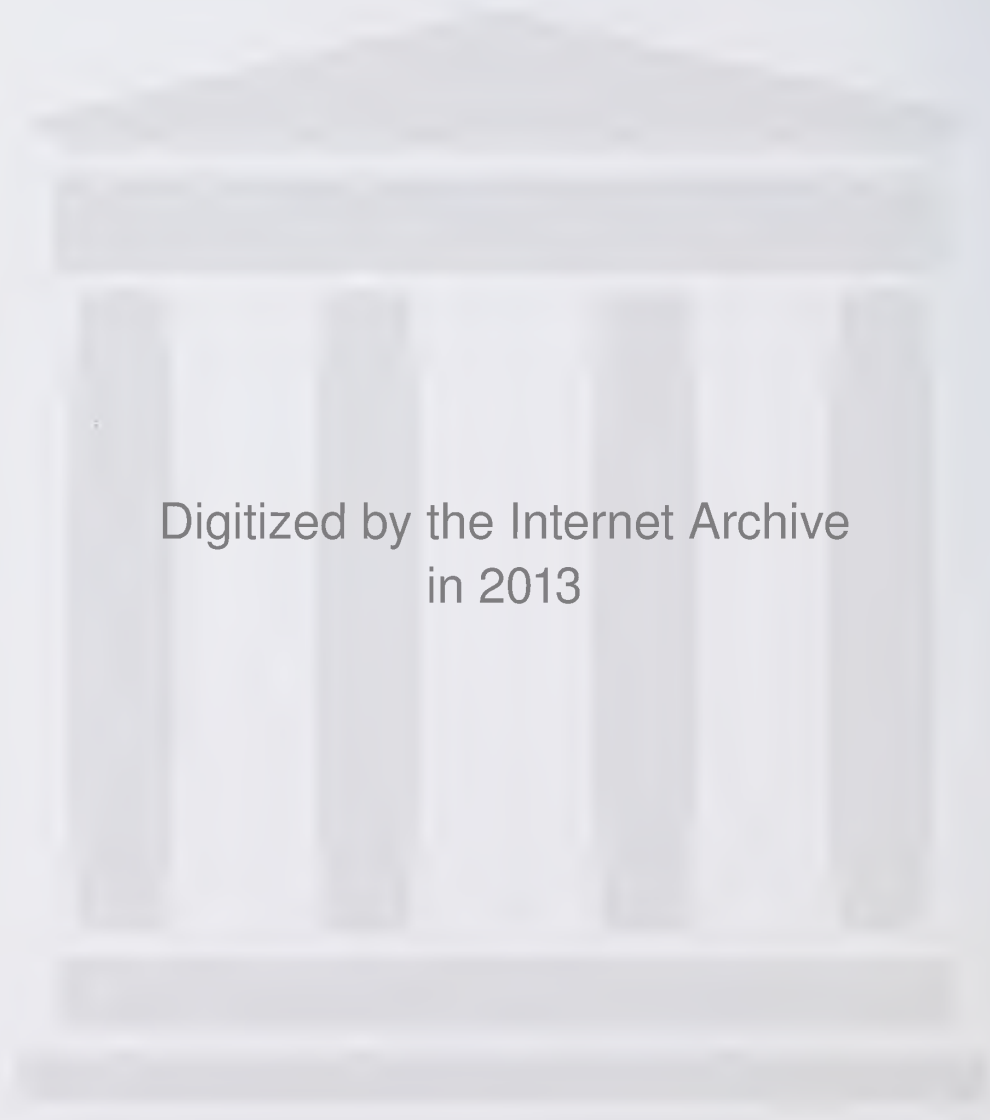
by

Michael John Pisterzi

June 1, 1970

Department of Computer Science
University of Illinois
Urbana, Illinois 61801

* This work was supported in part by the National Science Foundation under Grant No. US NSF GJ812 and Grant No. US NSF GJ813 and was submitted in partial fulfillment for the Doctor of Philosophy degree in Electrical Engineering, 1970.



ACKNOWLEDGEMENT

I would like to acknowledge the guidance, encouragement, and helpful suggestions of my advisor, Professor James E. Robertson. I would like to thank Mr. Webb T. Comfort for several suggestions that contributed significantly to the clarity of this paper.

I would also like to thank Miss Cheryl Becker for her skillful typing and the members of the Drafting Department of the Department of Computer Science for the figures that they drew.

Finally, I would like to thank my family, particularly my wife Candace, for their encouragement.

TABLE OF CONTENTS

Page

1. INTRODUCTION

1.1 Statement of the Problem 1
1.2 Relation to Prior Work 5
1.3 Structure of the Remainder of the Paper 7

2. INTRODUCTION TO THE METHOD OF PERFORMING THE PROCESSING

2.1 Organization of the Arithmetic Unit 11
2.2 Generalized Examples 22
2.3 The Basic Micro-Instruction Repertoire of the DPUs 26

3. THE ARITHMETIC CONSIDERATIONS OF IMPLEMENTING A LIMITED-CONNECTION ARITHMETIC UNIT

3.1 Introduction 35
3.2 Applicable Number Representations and Addition Methods 37
3.3 Multiplication Considerations 56
3.4 Multiplier Recoding 69
3.5 Normalization Considerations 75
3.6 Division Considerations 89

4. INTERACTION WITH MEMORY

4.1 Introduction 108
4.2 Methods Applicable When the Memory Byte is the Digit 109
4.3 Methods Applicable When the Memory Byte is a Number of Digits 114

5. OPERATIONAL SPECIFICATION OF THE MODULES

5.1 Introduction 123
5.2 The Digit Processing Unit 125
5.3 The Primitive Control Unit 136
5.4 The End Unit 140
5.5 Exponent Arithmetic Unit 142
5.6 The Sense Micro-Instruction Detector 143

Page

6. SUMMARY AND CONCLUSIONS

6.1 Discussion of Results 149
6.2 Suggestions for Related Work 152

APPENDIX

I. CHARACTERISTICS OF THE SYMMETRIC RADIX TWO SIGNED DIGIT ADDER 156

II. MINIMAL RIGHT-DIRECTED RECODER OF RADIX TWO SIGNED DIGIT NUMBERS 166

III. SELF-INITIALIZING MODULES 170

REFERENCES 171

VITA 174

A LIMITED CONNECTION ARITHMETIC UNIT

Michael John Pisterzi, Ph.D.
Department of Electrical Engineering
University of Illinois, 1970

A method of designing digital arithmetic units which are capable of performing floating point addition, subtraction, multiplication, division, and normalization is presented in this paper. The resulting arithmetic unit designs will be particularly appropriate for implementation in Large Scale Integration. The major characteristics of a limited connection arithmetic unit are:

1. It is composed of a large number of complex modules.
2. A very small number (three to eight) of specific module types are used.
3. The number of signal paths required between any module and the remainder of the arithmetic unit is small.
4. Each specific intermodule signal must be sent to a very small number of modules (one to three).

The paper shows that the complexity of the modules which are required to construct a limited connection arithmetic unit can be adjusted by selecting the number system and the details of the pro-

cessing algorithms. By this means the design can be tailored to the specific technology with which it is to be implemented.

The general arithmetic considerations of limited connection arithmetic units were also investigated. The major conclusion of this investigation is that signed digit number systems must be employed.

Several studies pertinent to radix two limited connection arithmetic units were also conducted. The probability of occurrence of each digit pair was determined for one of the three radix two adders. The results of this analysis were used in a study of normalization techniques. Finally, an optimum right-directed multiplier recoder was developed.

1. INTRODUCTION

1.1 Statement of the Problem

This study reports on a method of designing arithmetic units in a technology in which:

- the fan-out of the output signal from any logic element is limited,
- the basic building blocks are networks containing large numbers of logic elements,
- the number of signal paths through which a basic building block communicates to other basic building blocks is severely limited, and
- a small number of basic building block types must be employed.

These are, of course, characteristics of that class of technologies popularly known as Large Scale Integration, or LSI. The resulting arithmetic unit has some rather desirable properties not shared by other arithmetic unit organizations proposed for implementation in LSI (9, 10, 12)*.

The result of a given step of the calculation is determined digit by digit. Each digit of every result is determined in the

---

*Numbers in parentheses refer to articles listed in the References.

order in which it will be required when that result must be manipulated further, and it is stored in the same building block in which it will be needed when that result is to take part in additional processing. These properties of the arithmetic unit allow the processing of a second step to begin as soon as a sufficient (and small) number of digits of the results of the first step have been determined.

This paper presents a design procedure which has a high probability of evolving a design which effectively utilizes the potentialities of the technology. For the purposes of this paper, a given technology is considered to be effectively utilized when the average number of logic elements utilized per basic building block exceeds a fixed, predefined percentage of the number which can be constructed on it. The arithmetic unit will be developed as a large number of appropriately connected modules. These modules are functionally defined and are distinguished from basic building blocks so that we may postpone the question of whether a module can be implemented as one basic building block until after the modules are specified. We are able to do a significant amount of analysis without including specific technological considerations. The results of this analysis indicate that the major module type, the Digit Processing Unit (DPU), can be tailored to the technology by the selection of the

number representation employed by the arithmetic unit. The number of logic elements in the other principal building block, The Primitive Control Unit (PCU), is related to the details of the algorithms employed to perform multiplication, division and normalization, in addition to the number representation employed.

An arithmetic unit will consist of a large number of DPUs and only one PCU. If the performance objectives of the arithmetic units are not achieved by the design in which the DPUs and the PCU are implemented as single basic building blocks, more sophisticated algorithms may be employed to perform the basic operations. This would tend to increase the number of logic elements in the PCU without significantly affecting the design of the DPU. The PCU would then have to be implemented as several basic building blocks.

The study concentrates on the former problem; namely, the specification of module types required to construct an arithmetic unit. This problem is essentially independent of the technology with which the arithmetic unit is to be implemented. The results will show that the design has a sufficiently large number of parameters to allow this method of designing an arithmetic unit to be applied to any technology which has the characteristics mentioned earlier.

The design of the fractional part processor of a floating point arithmetic unit will be developed in detail in this paper. It will be capable of performing addition, subtraction, multiplication, division, and normalization. This emphasis on the fractional part processor is based on the relative number of digits and the processing required by the fractional part in contrast to that required by the exponent. The fractional part of a typical floating point number contains several times as many digits as the exponent does. In addition, the processing required by the fractional part of the operands is much more complex than that required by the exponent. Before an addition or subtraction is performed, the radix points of the operands must be aligned, which may require a number of right shifts. The result may also have to be shifted left to normalize it. In contrast, the exponents of the operands need only be subtracted to determine the number of radix alignment shifts required. The only other operation necessary is the addition of the number of normalization shifts to form the exponent of the result.

The difference in processing is even more extreme for multiplication and division. The fractional part calculation consists of a sequence of additions and shifts*, while the exponent

---

*This implies, of course, that there is only one adder in the fractional part processor. This assumption will be justified later.

calculation is basically a single addition.

Hence the processing required by the fractional part is more difficult than that required by the exponent. It is possible to evolve a design for the fractional part processor, as will be shown in this paper. The same design, or a simplified version of it may be employed for the exponent processor, although it may not be desirable to do so. The complete result of the exponent calculation is required to determine the processing to be performed on the fractional part, while only an estimate of the current results of the fractional part processing is required to determine what additional processing is necessary. For example, if an add is being performed, the difference of the exponents indicates the number of shifts required to align the radix points of the fractional parts, while only the first several digits of the sum of the fractional parts is necessary to determine if the sum is normalized. Hence, it may be desirable to perform the exponent calculations by a means which allows the entire results to be available in the shortest time.

1.2 Relation to Prior Work

The two known research efforts into the development of limited connection arithmetic units have evolved designs in which operations are performed by special purpose units (9, 10, 12).

The two efforts, while not identical, have several characteristics in common. The first of these is the development of special purpose units, namely the successful development of independent units to perform addition, subtraction, and multiplication. The second is the mode of operation. In both, operands travel through the arithmetic unit as a string of digits and must be gated to the appropriate execution unit. The pattern in which the basic building blocks are connected defines the processing that will be performed on the operands streaming through them. Finally, in both efforts control information is inherent in the logical structures of the mechanism.

The arithmetic unit developed in this paper is opposite in approach to those efforts. It has only one execution unit -- capable of addition, subtraction, multiplication, division, and normalization. The operands remain relatively stationary in this unit while being processed. The processing is controlled by control signals which propagate through the arithmetic unit.

This approach was taken because it holds promise of not requiring modifications to the programming systems employed on the computer. Computers having several specialized execution units (for the same class of operands) require either additional effort on the part of the programmer, more complicated compilers (1, 13, 22, 23), or special hardware (2, 24) to assure that all units

are kept busy as much of the time as practical while retaining the appropriate ordering of the operations where necessary.

This study evolves an arithmetic unit design in which all floating point operations* are performed by the same execution unit. The intent of this effort was to evolve an arithmetic unit that is operationally equivalent to the Von Neumann single adder arithmetic unit. Since there is one execution unit, there is no need for expending effort in attempting to maximize and coordinate the activity of independent execution units. Hence, the arithmetic unit developed here may be able to replace the arithmetic unit in existing computers with little or no change to the remainder of the computer and its programs. There is no need to develop specialized compilers; schemes similar to the Common Data Bus (24) are needed only in those systems which require buffering between the arithmetic unit and memory. The arithmetic unit described in this paper requires little or no additional development in other areas.

1.3 Structure of the Remainder of the Paper

In Chapter 2, the concept of the autonomous Digit Processing Unit is developed. The Digit Processing Units (or DPUs)

---

*Fixed point operations may also be performed by the unit.

each contain one digit of each of the active operands in its registers. The DPUs communicate with their neighbors in such a way that when one DPU is executing a given processing step (or micro-instruction), no other DPU is. The DPUs are organized so that the micro-instructions are passed from one DPU to the next in a specific order. The method of performing useful processing by causing the same sequence of micro-instructions to be performed by each of the DPUs will also be presented, as will the concept of micro-instructions streaming through the DPUs as they are executed by the DPUs in sequence. The chapter concludes by defining the micro-instructions which the DPUs must be able to perform and by indicating how these micro-instructions are combined to perform 'machine' instructions.

Chapter 3 treats the arithmetic aspects of the design. It discusses the number systems which can be employed, how addition overflow may be handled and how multiplication, division, and normalization may be performed. It also contains the development of a minimal right-directed recoder of radix two signed-digit numbers. The various possible methods of normalization are evaluated for radix two signed-digit numbers, and the optimum method is shown to depend on the ratio of shift time to the examination time.

Chapter 4 describes how the arithmetic unit and the memory containing its operands may communicate. The methods discussed are applicable when the memory byte* consists of an integral number of digits**. The methods discussed range from the very simple (entailing no additional equipment in the repetitive portion of the arithmetic unit) to the rather complex, employing a significant amount of special logic.

The operational description of the modules required to construct a limited connection arithmetic unit are described in Chapter 5. An attempt has been made to relate types of modules required and their characteristics with such parameters as the size of memory byte, the number system, and the algorithms for performing the machine instructions.

Conclusions and suggestions of related research are presented in Chapter 6. As indicated there, the attempt to construct a Limited Connection Arithmetic Unit in an appropriate

---

*Byte is used in this paper as the quantity of data which a device (e.g., the storage unit) operates on simultaneously. Byte is used in its arbitrary sense in this paper, which stresses that no assumption was made concerning the relative size of the data unit of the storage unit with respect to the sizes of digits and operands (which are denoted as words).

**In the unlikely event that several storage unit bytes are required to contain one digit, digit assembly/disassembly logic may be included in the DPUs or the PCU, and the method applicable to one digit per byte can be employed.

technology would be the best way of determining what additional considerations should be made.

Three appendices are included, which provide specific information required to design radix two arithmetic units. The first presents the analysis and results of a study of the steady state probabilities of each possible pair of digits in the representation of sums of the symmetric radix two signed-digit adder. The radix two minimal multiplier recoder developed in Chapter 3 is discussed further in Appendix II. A method of initializing the arithmetic unit that requires no additional connections is presented in Appendix III.

2. INTRODUCTION TO THE METHOD OF PERFORMING THE PROCESSING

2.1 Organization of the Arithmetic Unit

A basic Limited Connection Arithmetic Unit* consists of a Primitive Control Unit, a number of Digit Processing Units, and an End Unit. The Primitive Control Unit (PCU) receives instructions from some external device and converts them into a sequence of micro-instructions to be executed by the Digit Processing Units (DPUs). The conversion which the PCU performs is very similar to the conversion performed by the adder control logic of contemporary single adder arithmetic units. For example, a multiply is converted into a number of shifts and adds.

The DPUs collectively contain the fractional parts of all active operands and do the processing on them. The DPUs have the capability of performing micro-instructions which will (when performed by all DPUs) form sums, perform shifts, and do inter-register transfer.

The End Unit allows the last DPU to be identical to all the other DPUs and to operate as though it had a DPU on its right.

---

*Chapters 4 and 5 will discuss variations which employ additional modules.

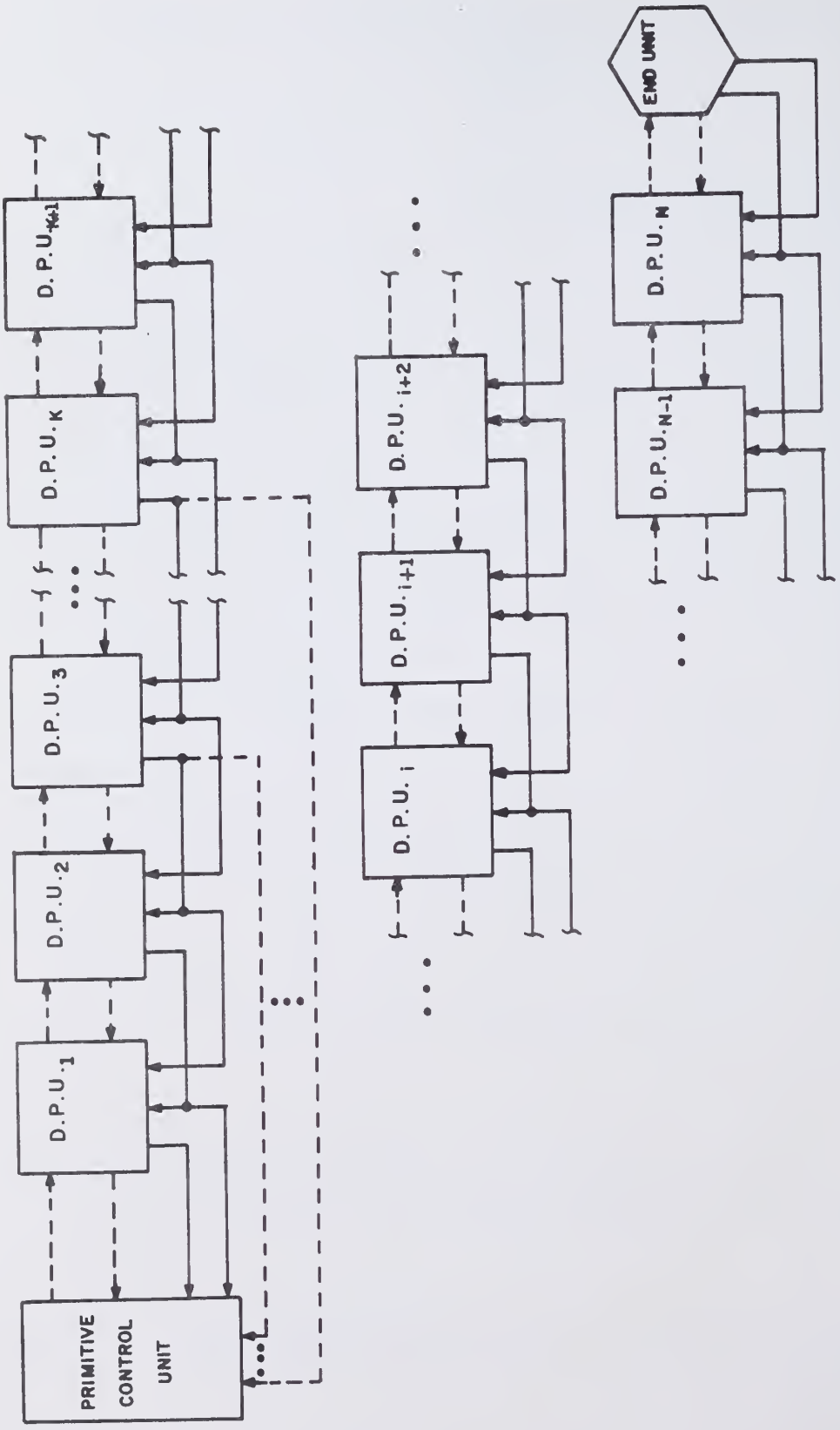
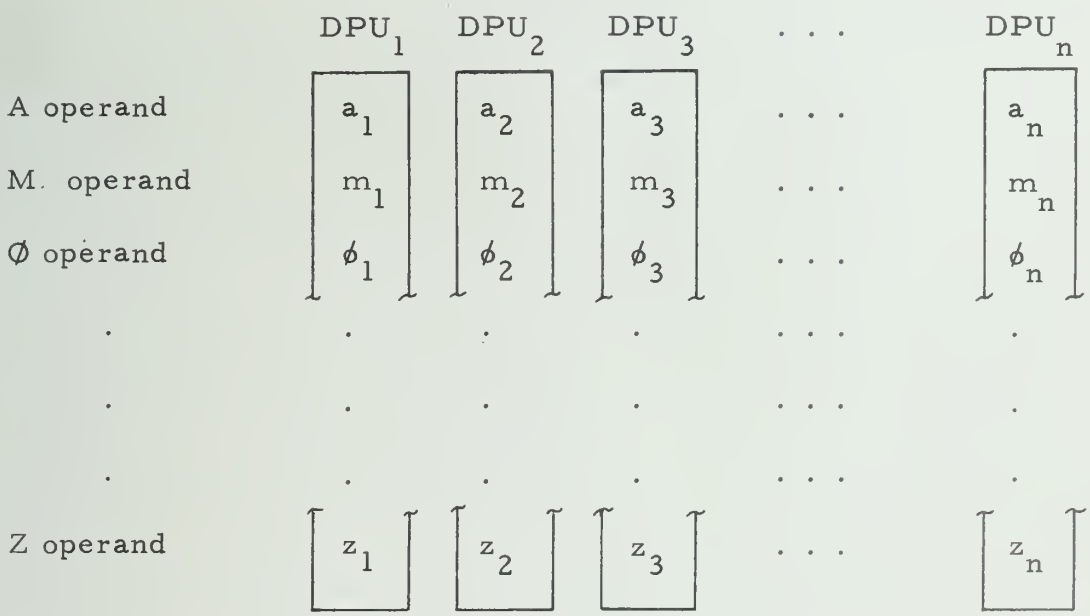


Figure 1. The organization of a typical limited connection arithmetic unit.



$$A = \sum_{i=1}^{n} a_i \, r^{-i}$$

$$M = \sum_{i=1}^{n} m_i \, r^{-i}$$

etc.

where r is an integer greater than one known as the radix of the arithmetic unit.

Figure 2 - The distribution of operands digits in the DPUs of a limited connection arithmetic unit.

The inter-module connections of a typical Limited Connection Arithmetic Unit is shown in Figure 1. Each DPU retains the values of one digit of each of the active operands in its register, as shown in Figure 2.

As mentioned earlier, each DPU performs the same sequence of micro-instructions*. From Figure 1 it can be seen that a given micro-instruction can not be executed by all DPUs in synchronization, but rather must be executed by them in sequence (i.e., first by DPU_1, then DPU_2, ...). As soon as all the DPUs which contain information required by DPU_1 to perform micro-instruction j+1 (referred to as μ_{j+1}) have executed μ_j and have sent the required information to DPU_1, μ_{j+1} may be performed by DPU_1. The micro-instructions are defined to have regular data requirements, so that as each additional DPU executes μ_j, one more DPU may execute μ_{j+1}. The micro-instructions may be viewed as flowing through successive DPUs.

This initial description brings out two characteristics that the set of micro-instructions is to have; namely, the need

---

*This is not absolutely necessary, as certain micro-instructions are performed to classify the value of an operand, as in selecting quotient digits. The DPUs not containing information required to make this classification need not perform these micro-instructions.

for information from as small a number of other DPUs as possible, and well-matched intrinsic execution rates. Both decrease the time between micro-instruction executions by DPU_1, and hence tend to yield high performance arithmetic units. This paper shows that the number of DPUs which must transmit information to a given DPU can be limited to one. The execution rate could not be taken into account at the level of this analysis. It is felt that differences in the execution rates of the micro-instructions can be minimized by employing more logic elements to perform the more complex (and hence potentially slower) micro-instructions.

The above description of the operation of the arithmetic unit indicates that the DPU registers do not contain entire operands as long as any of the DPUs are actively executing micro-instructions. Each DPU contains the digits of the results of the last micro-instruction it has executed. For example, if at a particular instant of time the following DPUs are active: DPU_3 (executing μ_{75}), DPU_5 (μ_{74}), DPU_8 (μ_{73}), DPU_{10} (μ_{72}), DPU_{15} (μ_{71}), . . . , the accumulator register distributed through the DPUs would contain:

$75a_1$, $75a_2$, (-), $74a_4$, (-), $73a_6$, $73a_7$, (-), $72a_9$, (-), $71a_{11}$, $71a_{12}$, $71a_{13}$, $71a_{14}$, (-), ... The dashes (-) indicate that the

corresponding digit cannot be explicitly identified as it may be changing. The number preceding the letter 'a' indicates which of the operands the given digit is part of. For example, ${}_{75}a_1$ is the value of the first digit of the 'A' register after the first seventy-five micro-instructions have been executed. This terminology will be used throughout this paper.

The processing performed by the DPUs can be described by the following:

$$ {}_j\vec{X}_i = \Psi_j ({}_{j-1}\vec{X}_i, {}_jF_{i-1}, {}_jG_{i+1}, \cdots {}_jG_{i+\alpha_j}) \qquad (2.1.1) $$

$$ {}_jF_i = \Phi_j ({}_{j-1}\vec{X}_i, {}_jF_{i-1}) \qquad (2.1.2) $$

and

$$ {}_jG_k = \Gamma_j ({}_{j-1}\vec{X}_k, {}_jF_{k-1}) \qquad (2.1.3) $$

where

${}_j\vec{X}_i$ is the operand information contained in the i^{th} DPU immediately following the execution of micro-instruction j. It is indicated to be a vector as it consists of the i^{th} digit of each of the active operands,

Ψ_j is the function employed to obtain the new operand set. It is dependent only on the micro-instruction to be performed,

${}_jF_i$ is a 'modifier' value which DPU_i transmits to DPU_{i+1} with the micro-instruction to be performed next,

Φ_j is the function which each DPU performs to determine ${}_jF_k$,

α_j is the number of DPUs which must cooperate with the DPU performing μ_j by transmitting to this 'active' DPU a value of ${}_jG_k$,

${}_jG_k$ is the value* which DPU_k transmits to the DPUs with which it cooperates when they are executing μ_j, and

Γ_j is the function all DPUs employ to determine the value of ${}_jG_k$ which it transmits to all DPUs with which it cooperates.

The operation of a typical DPU, DPU_i, is as follows. It begins in a state in which it is receptive to information defining

---

*The value of ${}_jG_k$ must be defined for $k \geq n+1$ where n is the number of DPUs in the arithmetic unit. One possibly is ${}_jG_k = 0$ for all $k \geq n+1$.

the next micro-instruction to be performed. DPU_i receives this information and the value of ${}_jF_{i-1}$ from its left neighbor, DPU_{i-1}. When DPU_i receives this information, it determines ${}_jG_i$ (i.e., performs Equation 2.1.3) and places this value on signal lines which are connected to several of its left neighbors. It also determines ${}_jF_i$ by performing Equation 2.1.2, and transmits this value and the identity of μ_j to DPU_{i+1}. Some time later DPU_i receives a signal from DPU_{i-1} indicating that DPU_{i-1} has executed μ_j. DPU_i then executes μ_j (i.e., performs Ψ_j), altering the value of one or more of its internal registers. DPU_i transmits a signal at this time to DPU_{i+1} which indicates that DPU_{i+1} may execute μ_j. When DPU_i receives an acknowledgment from DPU_{i+1}, it goes into the state where it is receptive to information concerning μ_{j+1}. The sequence above then repeats.

Notice that in this formulation the operations are completely independent of the significance of the digits retained by the DPU. All DPUs may then be identical. A second thing to note is that α_j, the number of DPUs to the right of the DPU performing the micro-instruction, is assumed to be a function of the micro-instruction being performed. Its value determines the desirability of including the micro-instruction in the repertoire of the DPUs. The larger the value of the parameter

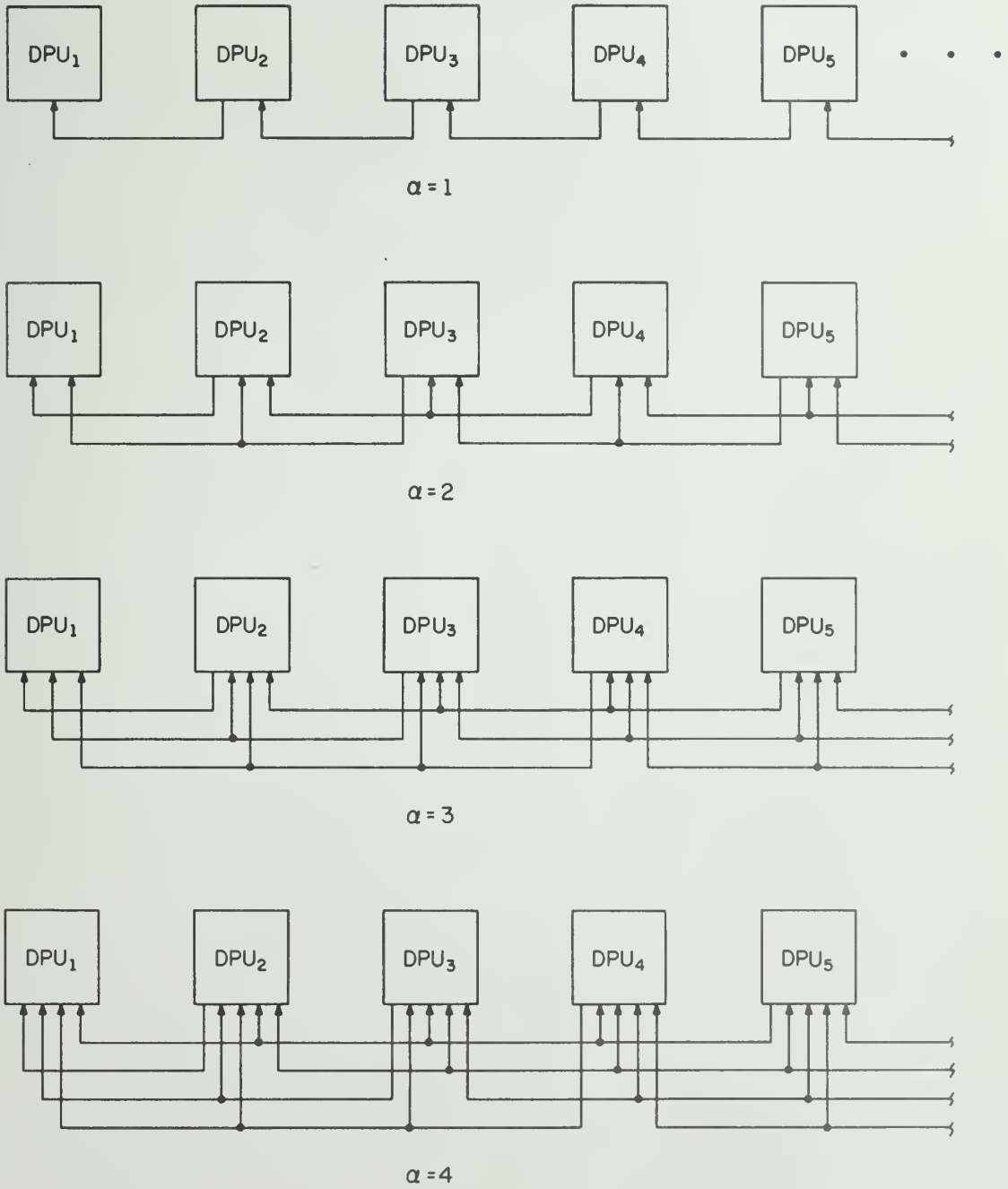


Figure 3. Inter-DPU data paths required for various values of α.

the less desirable the micro-instruction. There are two reasons for this. The first reason is that the subject micro-instruction cannot be performed by a DPU until the α_j DPUs to its right* have completed all previous micro-instructions. The execution rate of the subject micro-instruction is inversely proportional to α_j+1. The second reason is that the number of data paths required between the DPUs is determined by the maximum value of this parameter. This is shown in Figure 3, where α is the maximum of the α_j for the complete set of micro-instructions. Hence, the number of connections which must be made to a DPU is directly related to α. The communication of control information is not included specifically in the discussion above. Status information must be communicated from a given DPU to all DPUs to which it may supply data. Hence this communication network will also appear as shown in Figure 3.

A second item which must be minimized is the number of values taken on by ${}_jF_i$, the variable presented to DPU_{i+1} by DPU_i when it identifies the next (j^{th}) micro-instruction. This factor has a less profound effect on the data communications requirements. It affects only the number of transmission paths

---

*Or all the DPUs to its right, if there are less than α_j remaining in the sequence.

Table 1. α_j of the Micro-instruction of the Example.

j	1	2	3	4	5	6
α_j	2	1	0	1	2	0

Time	Micro-instruction					Operand Register				
	μ_1	μ_2	μ_3	μ_4	μ_5	O_1	O_2	O_3	O_4	O_5
1			S_1			0	0	0	0	0
2	1			S_1		1				
3	S_2	1			S_1		1			
4		S_2	1					1		
5	2		S_2	1		2			1	
6	3	2		S_2	1	3	2			1
7	S_4	3	2		S_2		3	2		
8		S_4	3	2				3	2	
9	4		S_4	3	2	4			3	2
10	S_5	4		S_4	3		4			3
11		S_5	4		S_4			4		
12			S_5	4					4	
13	5			S_5	4	5				4
14	6	5			S_5	6	5			
15		6	5				6	5		
16			6	5				6	5	
17				6	5				6	5
18					6					6

Figure 4. Example of the operation of a limited connection arithmetic unit.

necessary between adjacent DPUs over and above that required due to α.

2.2 Generalized Examples

An example of generalized operations will now be presented to illustrate that the processing of several micro-instructions may take place simultaneously in the arithmetic unit each by a different DPU. The α_j are given in Table 1. The arithmetic unit will have five DPUs and one operand. The operands will be indicated as A_j, the value of the operand after the j^{th} micro-instruction. This operand will be composed of five digits ${}_ja_1, \ldots, {}_ja_5$ such that digit ${}_ja_i$ is the digit contained in DPU_i after the j^{th} micro-instruction.

The operation of the arithmetic unit is presented in tabular form in Figure 4. The columns labelled O_i will indicate the operand contained in DPU_i. The occurrence of 'j' in the i^{th} operand column will indicate that ${}_ja_i$ has just been determined and placed in the operand register of DPU_i. The columns labelled μ_i will indicate when a micro-instruction has just been completed by DPU_i. The occurrence of 'j' in an instruction column will be used to indicate that the associated DPU has just executed micro-instruction j. The occurrence of 'S_j' in the i^{th} instruction column indicates that DPU_i has just received the

identity of μ_j and will begin determining ${}_jG_i$. The progression of time will be indicated by the rows, each row equivalent to the time required for a DPU to execute one micro-instruction*.

In Figure 4, the arithmetic unit is shown to be in a steady state at time 1. No micro-instructions are being executed and A_0 is in the operand register. We will assume that the identity of μ_1 has reached DPU_3 and ${}_1G_1$, ${}_1G_2$, and ${}_1G_3$ are available. At time 2, DPU_1 receives ${}_1G_2$ from DPU_2 and ${}_1G_3$ from DPU_3 and executes μ_1. This causes ${}_0a_1$ to be replaced by ${}_1a_1$. During the next four time intervals, μ_1 is performed consecutively by each of the remaining DPUs, since an additional ${}_1G_k$ becomes available just as it is required by a DPU to perform μ_1. The identity of the second micro-instruction, S_2, is received by a DPU one time unit after that DPU performs μ_1. Since DPU_1 requires ${}_2G_2$ to execute μ_2 ($\alpha_2 = 1$), this micro-instruction is not performed by DPU_1 until time 5, since it is not until that time that DPU_2 is able to determine this value and send it to DPU_1. Just as with μ_1, μ_2 is executed sequentially by each of the remaining DPUs during each of the next four time intervals.

---

*This presentation of the example is not intended to suggest that synchronous operation is necessary.

Micro-instruction 3 is performed by each of the DPUs one time unit after that DPU has performed μ_2 because $\alpha_3 = 0$, and no outside information ($_3G_k$) is required. The other micro-instructions are performed in the same pattern.

In general, DPU_i performs μ_j the time unit following its execution of μ_{j-1} if $\alpha_j = 0$; DPU_i performs μ_j after it has received $_jG_{\alpha_{j+i}}$ from DPU α_{j+i} if $\alpha_j \neq 0$.

Note that the number of DPUs in the arithmetic unit does not affect the rate of execution of micro-instructions.

To illustrate that the correct result is determined, the operation of the arithmetic unit is completed after executing six micro-instructions. The processing will then be completed by the end of time 18, at which time the contents of all the operand registers are indicated. The reader will note that their contents are $_6a_1$, $_6a_2$, $_6a_3$, $_6a_4$, and $_6a_5$, which constitutes A_6, as required.

If the six micro-instructions whose processing was depicted in Figure 4 were the micro-instructions required to accomplish one regular machine instruction, and are to be followed by other micro-instructions, the latter can clearly be initiated by DPU_1 at time $16 + \alpha_7$ (where μ_7 is the first micro-instruction for the next instruction). Hence, the micro-

instructions of successive instructions can be overlapped in the same manner that the micro-instructions of a given operation can. The time required to 'perform' an instruction is therefore not significantly affected by the number of DPUs comprising the arithmetic unit. The time required to perform an instruction is essentially the time between the performance by DPU_1 of the first micro-instruction of that instruction and its performance of the first micro-instruction of the next instruction*. This assumes, of course, that the time required for a typical micro-instruction to be performed by all of the other DPUs of the arithmetic unit is negligible in comparison to the time the arithmetic unit is busy executing a continuous string of micro-instructions**.

---

*For the last instruction, the time when a typical micro-instruction could be performed by DPU_1 should be used instead of the time the first micro-instruction of the next instruction is performed.

**This continuous string of micro-instructions is a result of the mapping of a continuous string of instructions into micro-instructions by the primitive control unit. The continuous string of instructions may be a result of the effective concatenation by a supervisory program of the instructions required by a sequence of tasks to be performed by the arithmetic unit. The time the arithmetic unit is busy on a continuous string of micro-instructions may then be on the order of several hours.

2.3 The Basic Micro-Instruction Repertoire of the DPUs

In this section we will discuss the micro-instructions which must be included in the repertoire of the DPUs so that the overall arithmetic unit is able to do addition, subtraction, multiplication, division, and normalization. The micro-instructions may be placed in four classes for the purposes of this discussion. These four classes are:

1. the inter-register transfers,
2. the shift micro-instructions,
3. the arithmetic micro-instructions, and
4. the memory accessing micro-instructions.

Micro-instructions in the first class cause operands to be transferred from one register to another. This allows the results of one machine instruction to be used as an operand in a subsequent instruction. For example, let us consider the case in which a third number is to be added to the quotient of a division that has just been performed. In additions, the number in the Ø register is added to the contents of the A register. Since the Ø register is used as the interface with the storage device, the contents of the MQ register (in this case the quotient of the division) must be transferred to the A register before the addition can be performed.

A second application of the inter-register transfer micro-instructions is in the exchange of operands when normalization or radix point alignment are required. If, like a classical Von-Neumann arithmetic unit, only the A register and the MQ register have the required shifting ability, an operand in the Ø register which must be shifted to align its radix point to that of the other operand must be moved to the A or MQ register.

In all the inter-register transfers, all of the data required by a DPU to perform the micro-instruction is contained within that DPU. This can be seen in Figure 2. Each DPU contains one digit of each of the operands. Therefore, $\alpha = 0$ for all inter-register transfers and ${}_{j}F_{i}$ is not required to transmit data. The value of ${}_{j}F_{i}$ may be used, therefore, to identify one or both of the registers taking part in the transfer. The number of micro-instruction codes which must be assigned to inter-register transfer micro-instructions is therefore dependent on the number of values which may be taken on by ${}_{j}F_{i}$ in addition to the number of pairs of registers between which inter-register transfers are to be performed.

In the notation of Equation 2.1.1 through 2.1.3 the inter-register transfer micro-instructions may be formulated as:

$$ {}_{j}x_{i} = {}_{j-1}y_{i} \qquad i = 1, 2, \ldots, n \qquad (2.3.1) $$

$$ {}_{j}F_{i} = {}_{j}F_{i-1} \qquad i = 1, 2, \ldots, n \qquad (2.3.2) $$

$$ {}_{j}G_{i} = \langle \text{null} \rangle \qquad i = 1, 2, \ldots, n \qquad (2.3.3) $$

where

Y	is the register to be copied into the X register,
${}_{j}x_{i}$	is the i^{th} digit of the X register after the transfer,
${}_{j-1}y_{i}$	is the i^{th} digit of the Y register before the transfer, and
$\langle \text{null} \rangle$	indicates that the value of ${}_{j}G_{i}$ is not required when performing inter-register transfers.

The second class of micro-instructions is the shift micro-instructions. They are used during radix point alignment prior to addition or subtraction, for normalization, and for multiplication and division by the radix during the repetitive steps for multiplication and division. We will assume that shifts of more than

one digital position will be performed as a number of successive shifts of one digital position each.

The left shift can be accomplished by causing the DPU to the immediate right of the DPU performing the micro-instruction to transmit the value of the digit of the operand contained in its register to the DPU performing the micro-instruction. This DPU stores the digit it receives in its operand register. The equations defining a left shift micro-instruction are:

$$ {}_{j}x_{i} = {}_{j}G_{i+1} \qquad i = 1, 2, \ldots, n \qquad (2.3.4) $$

$$ {}_{j}F_{i} = {}_{j}F_{i-1} \qquad i = 1, 2, \ldots, n \qquad (2.3.5) $$

$$ {}_{j}G_{i} = {}_{j-1}x_{i} \qquad i = 1, 2, \ldots, n \qquad (2.3.6) $$

$$ {}_{j}G_{n+1} = {}_{j}F_{n} \qquad \text{if } {}_{j}F_{n} \text{ is a valid digit} \qquad (2.3.7) $$

otherwise see text

where

X is the operand being shifted,

${}_{j}x_{i}$ is the i^{th} digit of the shifted operand,

$_{j-1}x_i$ is the i^{th} digit of X before the shift,

$_jF_i$ is the modifier value passed along with the micro-instruction.

$_jF_0$ is the value that the PCU sends to DPU_1 with the left shift micro-instruction to indicate the value that is to go into the last DPU. If $_jF_0$ is a valid digit, it becomes the digit shifted into the last DPU. If it is not a valid digit, it causes the End Unit to shift in the digit shifted out during the last right shift.

One should also note that the left shift micro-instructions make it possible to transmit the most significant digit of an operand to the PCU. The value of this digit will be on the $_jG_1$ lines just prior to the execution of the left shift micro-instruction by DPU_1. The left shift can therefore be used by the PCU to examine operands.

The right shift micro-instruction does not have the complexity of the left shift micro-instruction. The value stored into a DPU is the value transmitted by its left neighbor DPU with the indication that a right shift is to be performed. The value of the digit stored in the first DPU is determined

by the PCU. In the terminology of Equations 2.1.1 through 2.1.3,

$$ {}_{j}x_{i} = {}_{j}F_{i-1} \qquad i = 1, 2, \ldots, n \qquad (2.3.8) $$

$$ {}_{j}F_{i} = {}_{j-1}x_{i} \qquad i = 1, 2, \ldots, n \qquad (2.3.9) $$

$$ {}_{j}G_{i} = \langle \text{null} \rangle \qquad i = 1, 2, \ldots, n \qquad (2.3.10) $$

where

${}_{j}F_{0}$ is the digit which the PCU transmits with the indication that a right shift is to be performed. This value becomes the value of the most significant digit of the shifted operand.

The value of ${}_{j}F_{n}$, which is transmitted by DPU_{n} to the 'End Unit', where it is stored as the new top element in the push-down stack.

A final note concerning shifts is that the value of $\alpha_{LS}=1$ and $\alpha_{RS}=0$; and that in the worst case (left shift), ${}_{j}F_{i}$ must take on one value more than the number of values a digit may assume. Two micro-instruction codes are required for each register which has shifting capabilities.

The third class of micro-instructions are those micro-instructions required to control the arithmetic processing of the operands. Multiplication and division will be implemented as a number of additions and shifts, just as they were in the classical Von-Neumann arithmetic unit. The only micro-instructions necessary in this case are those which cause A to be replaced by the following expression

$$A_j = A_{j-1} + (k * \emptyset_j) \qquad (2.3.11)$$

where

A_j is the value of the operand contained in the A register after the arithmetic operation,

A_{j-1} is the value of the operand contained in the A register prior to the arithmetic operation,

$\emptyset_j$ is the value of the operand contained in the $\emptyset$ register, and

k is a number whose magnitude is less than the radix employed by the arithmetic unit.

It is shown in Section 3.1 that k does not take on values which are not digits of the number representation. The ${}_jF_i$ interconnections may therefore be employed to distribute the value of k to the DPUs with no need to increase the number of

interconnections, since the ${}_jF_i$ data paths must convey all possible digit values for the shifting micro-instruction.

In Section 3.2 we show that the addition function can be implemented such that α is 1 or α is 2. In the implementations for which α is 1, either the radix and the values which may be taken on by k are restricted, or two micro-instructions must be performed to complete each addition. In the implementation for which α is 2, no such restrictions are encountered. The choice of which method of addition is implemented must be based on trade-off considerations. The major points to be considered are:

- The average time taken to perform additions by the three possible methods.
- The cost of implementing an arithmetic unit whose DPUs have $\alpha = 2$ compared to the cost if $\alpha = 1$. This cost has three components. The first component is the larger number of interconnections which must be made. In addition to transmitting ${}_jG_i$ and receiving ${}_jG_{i+1}$, DPU_i must be able to receive ${}_jG_{i+2}$ when $\alpha = 2$. The second component of this cost is the additional logic elements required because of the larger number of signals it must receive, the more complex control

logic, and the more complex addition logic. The third component is the increased time to perform all micro-instructions; the control logic in the DPU is more complex when $\alpha = 2$ and, quite likely, slower.

The last class of micro-instructions are those which cause exchanges of data between the arithmetic unit and the storage unit. There will be no micro-instructions in this class if the PCU acts as an intermediary in the exchange as suggested by Comfort (8). The micro-instructions in this class are similar to the inter-register transfer micro-instructions in that ${}_{j}F_{i}$ may be used to identify (or aid in identifying) the register taking part in the exchange and the type of exchange (i.e., load or store). The number of micro-instruction codes required is related to the number of registers which communicate with the storage unit, and to the number of values which ${}_{j}F_{i}$ may take on. Interfacing the storage unit to the arithmetic unit is discussed in depth in Chapter 4.

3. THE ARITHMETIC CONSIDERATIONS OF IMPLEMENTING A LIMITED-CONNECTION ARITHMETIC UNIT

3.1 Introduction

The limited connection arithmetic unit is organized to process floating point operands. The fractional part of the operands will be contained in, and processed by, the DPUs. The processing will begin with the most significant digits of the operands and proceed to those with decreasing significance. This is necessitated by the requirements of the normalization and division processes. In both of these processes, the values of the most significant digits of the operands determine what additional processing is required. In normalization, the value of the most significant digits are examined to determine if the number has been normalized, and if not, the next step to be taken. During division, the approximate value of the partial remainder, determined by examining several of the most significant digits, is used in the selection of the next quotient digits. In both of these cases, the results of examining several of the most significant digits of the operands determines the next several micro-instructions to be performed, and hence must be performed in as short a time as possible. This then makes it necessary for the most significant digits of the operands

to be placed in the DPU which is adjacent to the PCU, since it is the first DPU to execute each micro-instruction.

The various methods of implementing the addition micro-instruction (see Equation 2.3.11) are discussed in Section 3.2, together with a discussion on the implications of each method on the complexity and performance of the arithmetic unit. Section 3.3 discusses the overflow recoder which must be implemented in the PCU if the arithmetic is to be able to form double length product fractions. The desirability of recoding multiplier digits is discussed in Section 3.4. A class of optimum recodings for radix 2 signed-digit numbers is also developed in this section. Normalization is discussed in Section 3.5. The optimum algorithm for normalizing radix 2 signed-digit numbers is determined; it is shown that the optimum algorithm for a specific arithmetic unit is dependent on the value of ξ of that arithmetic unit*. Division is analyzed in Section 3.6. The relationship between the parameters of the number system and the number of quotient digits determined during each examination of the partial remainder is determined in this section. This analysis shows that it is possible to imple-

---

*The ratio of the time for the PCU to sense the value of the first two digits of the number normalized to the time to shift that number one digital position to the left.

ment division without requiring special data paths if one quotient digit is determined during each examination of the partial remainder and if an $\alpha = 2$ adder is implemented.

3.2 Applicable Number Representations and Addition Methods

In order to perform multiplication and division by alternately doing one addition and shifting, the arithmetic unit must include micro-instructions to add or subtract various multiples of one of the operands to the other. That is, the arithmetic micro-instructions must be characterized as follows:

$$A' = A + (k*\emptyset) \tag{3.2.1}$$

where

A', A, $\emptyset$ are consistently represented numbers, and

k is a multiplier or quotient digit such that

$$|k| \leq K, \qquad \frac{r-1}{2} \leq K \leq r-1. \tag{3.2.2}$$

Because of the necessity of beginning addition and subtraction with the most significant digit and with limited information concerning values of the operands, the conventional, non-redundant number representations cannot be used. Three

methods of performing addition on signed-digit numbers were investigated. Because it must be possible for the sum of one addition to be used as an operand of a subsequent addition, the Avizienis Adder and the Second Order Simple Transformation Adder were found to be equivalent. The designer has three methods of implementing addition, in the Limited Connection Arithmetic Unit, two of which have $\alpha = 1$, the other which has $\alpha = 2$.

The Avizienis Adder requires that the α of the arithmetic unit to be at least one. The number of addition micro-instructions which must be issued to perform one addition operation, as defined by Equation (3.2.1) and (3.2.2), is determined by the value of k. The number of micro-instructions which must be issued varies from one to three. Furthermore, the Avizienis Adder cannot be used on radix two numbers.

The two remaining methods employ the Three Level Adder. An adder of this type may be designed for any signed-digit number system. The first of these methods is a straightforward implementation and requires that the α of the arithmetic unit be at least two. Only one micro-instruction must be issued for each addition operation. The second of these methods implements the Three Level Adder as a two step process, and hence

requires two micro-instructions to be executed for each arithmetic operation. This method requires that the $\alpha\alpha$ of the arithmetic unit be at least one. It has one major drawback over the other methods: the accumulator register must be able to assume approximately $\frac{r}{2}$ times as many states as it must for either of the other methods.

3.2.1 The Avizienis Adder

The Avizienis Signed Digit Adder (4, 5, 6) can be characterized in the notation of Equations (2.1.1) through (2.1.3) by $\alpha = 1$. That is, a digit of sums and differences can be formed based on the operand digits of that digital position and those of the digital position to its "right". The arithmetic operations as proposed by Avizienis and these number representations cannot satisfy the condition defined by Equations (3.2.1) and (3.2.2) but can be characterized by $K \leq \frac{1}{2}(r-1)$ (5). Avizienis suggests that either an odd radix be employed and the multiplier digits be recoded to meet the restriction on the value of k, or that two arithmetic operations be performed between shifts when k exceeds the range. The first of these is not applicable to this structure as it requires that the recoder yield a non-redundant representation of the operand. This is not possible because the recoder must begin at the most significant

digital position, precluding a non-redundant output. However, the second approach is applicable; the arithmetic operations can be completed in two additions for any value of K if an odd radix is employed. However, if an even radix is used, $|k| = r-1$ would require three additions. It seems clear that the relative frequency with which the adder circuitry must be used twice between shift operations can be reduced. Also, the possibility that the adder must be used three times can be eliminated by an appropriate recoding of multipliers and choice of division parameters. An additional restriction on the use of this scheme is that the radix must be greater than two.

3.2.2 The Second Order Simple Transformation Adder

Let us now consider an alternative addition scheme proposed by Rohatsch (21). This method requires the values of one additional digit of the operands (i.e., $\alpha = 2$). It is described as the second order simple transformation by Rohatsch, and is like the Avizienis structure in having two levels. Rather than two intermediate digits, this structure employs three, with relative weights of r^2, r, and 1 for the f's, t's, and w's respectively (see Figure 5).

The operand digits (a'_i, a_i, ϕ_i) are all assumed to be

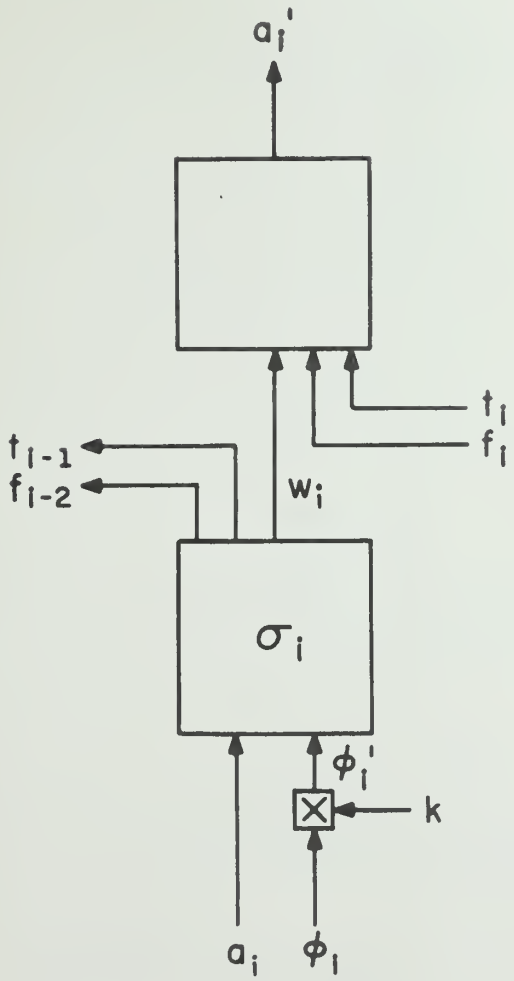


Figure 5. The two level adder based on the second order simple transformation.

chosen from the same digit set for reasons of compatibility. That is, if

$$B_i = \{i, (i-1), \ldots, 1, 0, \bar{1}, \ldots \bar{i}\} \quad i \geq 1 \qquad (3.2.3)$$

where an overbar indicates negation,

then

$$a_i \in B_{r-\ell} \text{ , } a'_i \in B_{r-\ell} \text{ , and } \phi_i \in B_{r-\ell} \qquad (3.2.4)$$

The multiplier digit k may be selected either from the set $B_{r-\ell}$ above or the set B_{r-1}. The latter option is included as a possibility because it may be desirable to recode multiplier digits in an attempt to minimize the number of uses of the adder, and it may be desirable to conduct the division with a more redundant number representation so as to decrease the number of digits of each partial remainder which must be examined (3). In the latter case, the quotient digits thus obtained must be applied to a recoder which converts them to a compatible representation. We will investigate adders compatible with both constraints on the multiplier digits.

The equations describing the adder of Figure 5 are

$$\sigma_i = a_i + k\,\phi_i = f_{i-2}\, r^2 + t_{i-1}\, r + w_i \qquad (3.2.5)$$

and

$$a'_i = f_i + t_i + w_i \qquad (3.2.6)$$

where

a_i, ϕ_i are the i^{th} digits of the operands, as discussed above

a'_i is the i^{th} digit of a representation of $A + k * \phi$.

σ_i is the i^{th} digit of an intermediate representation of $A + k * \phi$.

f_{i-2} is one of the components into which σ_i is recoded.

It has weight r^2.

t_{i-1} is another of the components into which σ_i is recoded.

It has weight r.

w_i is the last of the components into which σ_i is recoded.

It has weight 1.

Using the analysis of Rohatsch,

$$S(T) \geq r \qquad (3.2.7)$$

$$S(W) \geq r \qquad (3.2.8)$$

is a necessary condition for $r^2 F + rT + W$ to be contiguous,

and hence "cover" Σ , where

F, T, W, and Σ are digit sets such that

$f_i \in F$, $t_i \in T$, $w_i \in W$, and $\sigma_i \in \Sigma$, and

S(X) is the number of elements in the set X.

Rohatsch has shown that the choice of equality in (3.2.7) and (3.2.8) above allows the size of Σ to be maximum since the size of A' is fixed. Making this choice and applying it to the condition that F + T + W is covered by A',

$$S(A') \geq 2r + S(F) - 2.$$

But from (3.2.3) and (3.2.4) we see that

$$S(A') = 2(r - \ell) + 1,$$

so

$$S(F) \leq 3 - 2\ell .$$

But since

$$\ell \geq 1,$$

then

$$S(F) \leq 1. \qquad (3.2.9)$$

Note that if S(F) = 1, the f input of the upper stage is constant, and the structure therefore degenerates to the Avizienis structure. The conclusion is that it is not possible to devise a two level adder such that the conditions of Equations (3.2.1) and (3.2.2) can be met.

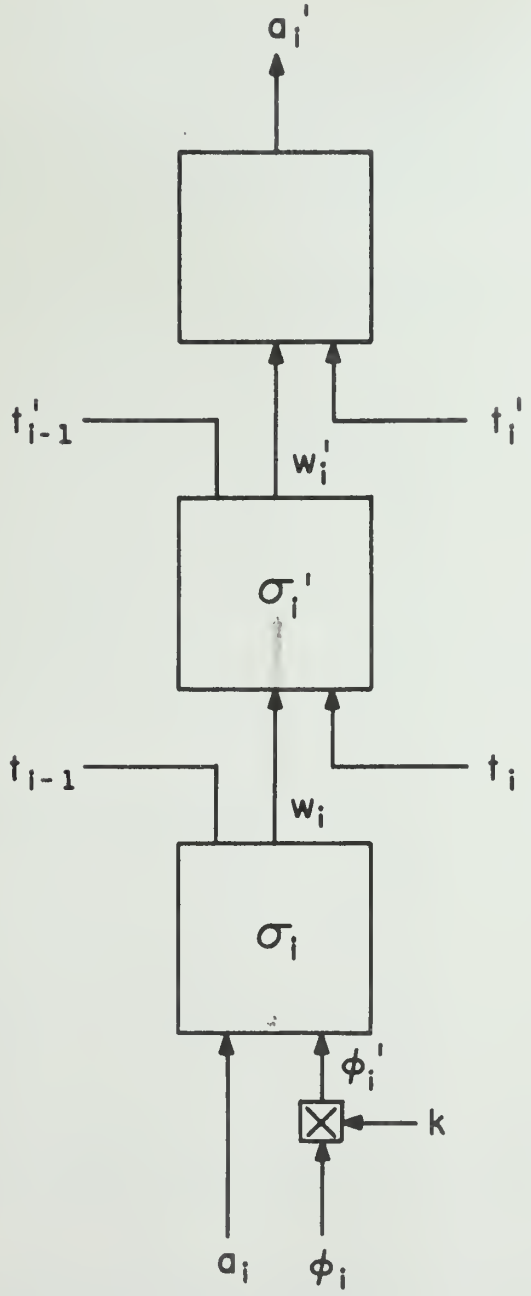


Figure 6. The three level adder based on the application of two simple transformations.

3.2.3 The Three Level Adder

We will now analyze a three level structure which can also be characterized by $\alpha = 2$. The notation for this adder, shown in Figure 6, is as follows:

$$\sigma_i = a_i + k \phi_i \qquad (3.2.10)$$

$$w_i = \sigma_i - t_{i-1} r \qquad (3.2.11)$$

$$\sigma'_i = t_i + w_i \qquad (3.2.12)$$

$$w'_i = \sigma'_i - t'_{i-1} r \qquad (3.2.13)$$

$$a'_i = t'_i + w'_i \qquad (3.2.14)$$

where

a_i, ϕ_i are the i^{th} digits of the operands

a'_i is the i^{th} digit of a representation of $A + k * \phi$

σ_i, σ'_i are the i^{th} digits of intermediate representation of $A + k * \phi$.

t_i, w_i, are the components into which σ_i is recoded.

t'_i, w'_i, are the components into which σ'_i is recoded.

In the analysis below, the sets from which the digits are chosen are as follows:

$$a_i \in A,\ a'_i \in A',\ \phi_i \in \phi,\ k \in K,\ \sigma_i \in \Sigma,$$

$$\sigma'_i \in \Sigma',\ t_i \in T,\ t'_i \in T',\ w_i \in W,\ w'_i \in W'.$$

Equation (3.2.14) yields

$$S(A') \leq S(T') + S(W') - 1.$$

Using (3.2.3), (3.2.4), and assuming that

$$S(W') = r,$$

since this maximizes* the set Σ' covered by T' + W', the equation becomes an equality

$$2(r-\ell) + 1 = S(T') + r-1,$$

or

$$S(T') = r-2\ell + 2. \qquad (3.2.15)$$

Since the size of T' must be 2 or greater to assure that the three level adder does not degenerate into the Avizienis adder,

$$r - 2\ell + 2 \geq 2,$$

or

$$r \geq 2\ell .$$

This condition is satisfied for all number systems considered in this paper.

Now, from (3.2.13) and (3.2.15) the size of Σ' is

$$S(\Sigma') = S(T') \cdot S(W') = r(r-2\ell + 2). \qquad (3.2.16)$$

From Equation (3.2.12) we see that

$$S(\Sigma') \leq S(T) + S(W) - 1$$

---

*Since A' has a fixed size.

Using (3.2.16) and assuming that

$$S(W) = r,$$

since this maximizes* the size of Σ covered by T and W and allows the equation above to become the following equality

$$S(\Sigma') = S(T) + r - 1.$$

Applying (3.2.16) to the above yields

$$S(T) = r^2 - 2r\ell + r + 1 \qquad (3.2.17)$$

From Equations (3.2.10) and (3.2.11) we see that

$$S(T) \cdot S(W) \geq S(\Sigma') \geq S(A) + S(k \cdot \emptyset) - 1.$$

Applying (3.2.17) and the assumption that S(W) = r, this becomes

$$r^3 - 2r^2\ell + r^2 - r \geq 2(K + 1)(r - \ell) + 1 \qquad (3.2.18)$$

Setting K = r-1, which is equivalent to assuming that $k \in B_{r-1}$, (3.2.18) reduces to

$$\ell \leq \frac{r}{2} + \frac{1}{2r} \qquad \text{when } r > 1 \qquad (3.2.19).$$

Since $\ell \leq \frac{r}{2}$ for all signed-digit number systems, it is possible to implement a three level adder for any signed-digit number system. This conclusion is valid whenever all multiplier digits k are chosen from B_{r-1} or one of its subsets. Hence, the redundancy of multiplier and quotient digits has only a second-order effect on the adder complexity. The number of DPUs to

---

*Since the size of Σ' is fixed by our earlier assumption.

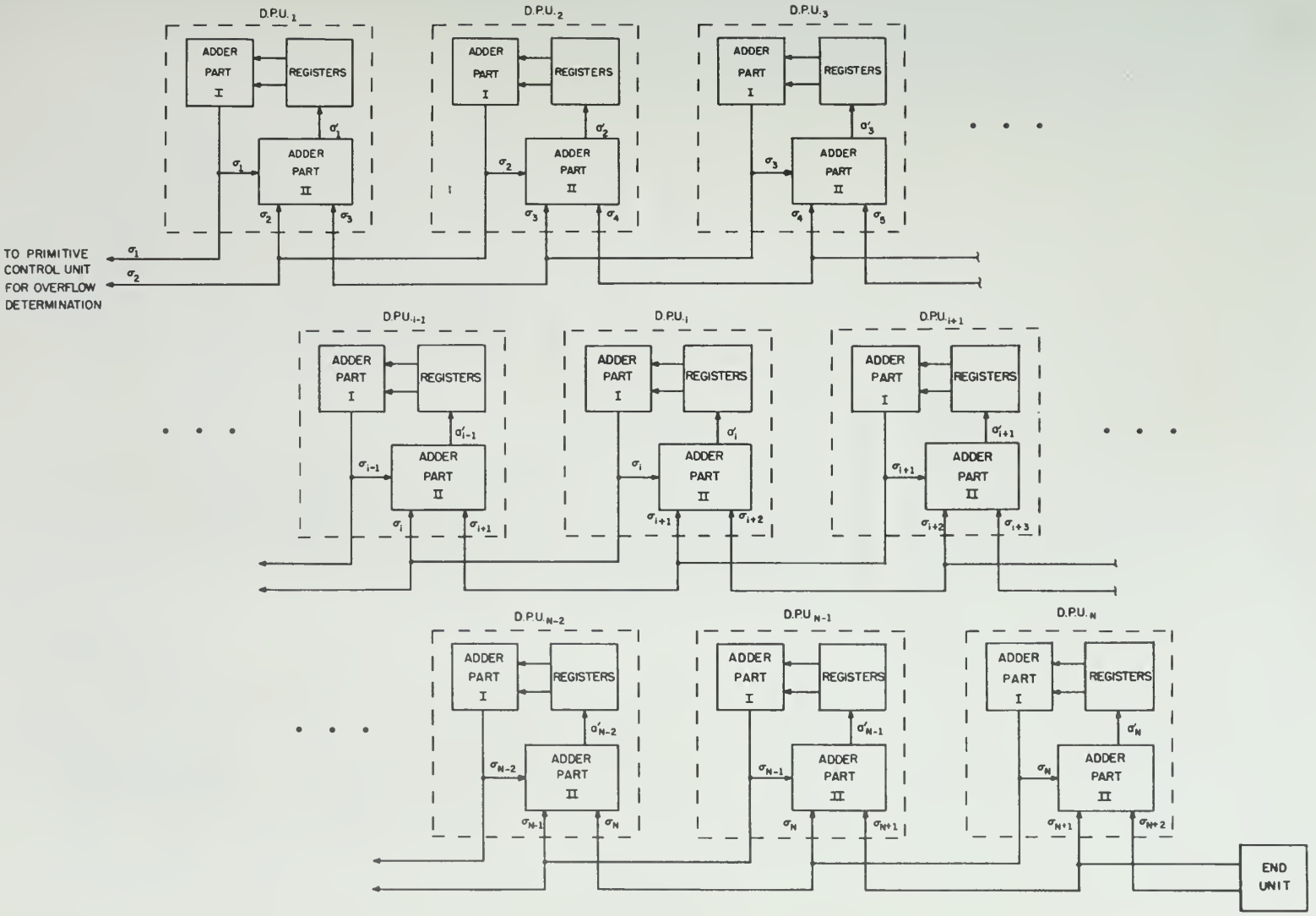


Figure 7. Implementation of the three level adder that requires one micro-instruction per addition.

which a DPU must send information for performing additions is independent of the range of multiplier digits employed, while the amount of information is dependent on this factor.

3.2.4 Implementing the Three Level Adder

The designer has two methods of implementing the three level adder in the limited connection arithmetic unit. The first method is to implement it so that one micro-instruction is executed for each addition operation. If this method is chosen, the data processing portion of the structure can be represented schematically as in Figure 7. The registers, of course, retain one digit of each of the operands. Adder Part I determines the appropriate σ_i from the data retained in the registers of the Digit Processing Unit and the value of k, which is carried by the control (micro-instruction) stream. That is, each Adder Part I implements Equation 3.2.10. It transmits this information to the Adder Part II of its own DPU, and to those of the two DPUs to its left. Each Adder Part II then determines its appropriate a'_i digit based on the values of the σ's presented to it. That is, the transfer function of Adder Part II is the combined transformations of Equations 3.2.11 through 3.2.14.

Note that in this case, the α of the arithmetic unit must be two or more. A second method allows α to be one by

requiring two micro-instructions to be executed for each addition operation. Equations (3.2.10) through (3.2.12) must be performed by the first micro-instruction, which causes an interim representation, Σ', of the sum to be placed in the accumulator register. The second micro-instruction then recodes this interim representation to an acceptable signed-digit number by performing Equations (3.2.13) and (3.2.14).

This method of implementing addition is subtly different from the Avizienis Adder, described in Section 3.2.1. From one to three uses of the Avizienis Adder are required to perform an addition operation, while two addition micro-instructions are required in the method just described. Moreover, this method of performing addition requires that the number of states taken on by a digit of the accumulator be larger than that required in the Avizienis adder by approximately a factor of $\frac{r}{2}$, the radix of the number system employed (see Equation 3.2.16). Hence, if the arithmetic unit must be implemented such that $\alpha=1$, the Avizienis Adder appears to be a more optimum design than the two step implementation of the three level adder.

3.2.5 A Detailed Look at the Radix Two Three Level Adder

We will now discuss the simplest of the adders of the above type, the radix two three level adder. It is presented

as both an example and a means of examining in greater detail the requirements of the adder.

In the case of radix two, K =1 and ℓ must be chosen to be one. There are three possible adders. The first, which will be referred to as Adder 1, can be characterized by

$$T = (1, 0, \bar{1}),\ W=T' = (0, \bar{1}),\ \text{and}\ W' = (1,0).$$

The second adder, which will be referred to as Adder 2, can be characterized by

$$T = (1, 0, \bar{1}),\ W = T' = (1, 0),\ \text{and}\ W' = (0, \bar{1}).$$

The third adder, Adder 3, can be characterized by

$$T = W = T' = W' = (1, 0, \bar{1}).$$

The operands for all adders are

$$A = \sum_{i=1}^{n} a_i 2^{-i} \qquad \text{and } \emptyset = \sum_{i=1}^{n} \phi_i 2^{-i},$$

the sum is

$$A' = \sum_{i=0}^{n} a'_i 2^{-i}.$$

The relationships between them are given by Equations (3.2.10) through (3.2.14), in which r = 2 and $|k| = 1$. The choices in Equation (3.2.11) are given for all values of σ_i, and for all three adders in Table 2.

Table 2. Digit Choices in (3.2.11) For The Radix 2 Adder Types

Input	Adder 1		Adder 2		Adder 3	
σ_i	t_{i-1}	w_i	t_{i-1}	w_i	t_{i-1}	w_i
2	1	0	1	0	1	0
1	1	$\bar{1}$	0	1	1	$\bar{1}$
0	0	0	0	0	0	0
$\bar{1}$	0	$\bar{1}$	$\bar{1}$	1	$\bar{1}$	1
$\bar{2}$	$\bar{1}$	0	$\bar{1}$	0	$\bar{1}$	0

Table 3. Digit Choices in (3.2.13) For The Radix 2 Adder Types

Input	Adder 1		Adder 2		Adder 3	
σ'_i	t'_{i-1}	w'_i	t'_{i-1}	w'_i	t'_{i-1}	w'_i
2	_	_	1	0	1	0
1	0	1	1	$\bar{1}$	0	1
0	0	0	0	0	0	0
$\bar{1}$	$\bar{1}$	1	0	$\bar{1}$	0	$\bar{1}$
$\bar{2}$	$\bar{1}$	0	_	_	$\bar{1}$	0

Table 4. Combined Adder Tables.

		Adder 1						Adder 2						Adder 3					
		σ_i					a'_0	σ_i					a'_0	σ_i					a'_0
σ_{i+1}	σ_{i+2}	2	1	0	$\bar{1}$	$\bar{2}$	i=0	2	1	0	$\bar{1}$	$\bar{2}$	i=0	2	1	0	$\bar{1}$	$\bar{2}$	i=0
2	2	1	0	1	0	1	1	0	1	0	1	0	2	1	0	1	0	1	1
2	1	1	0	1	0	1	1	$\bar{1}$	0	$\bar{1}$	0	$\bar{1}$	1	1	0	1	0	1	1
2	0	1	0	1	0	1	1	$\bar{1}$	0	$\bar{1}$	0	$\bar{1}$	1	1	0	1	0	1	1
2	$\bar{1}$	1	0	1	0	1	1	$\bar{1}$	0	$\bar{1}$	0	$\bar{1}$	1	1	0	1	0	1	1
2	$\bar{2}$	0	$\bar{1}$	0	$\bar{1}$	0	0	$\bar{1}$	0	$\bar{1}$	0	$\bar{1}$	1	1	0	1	0	1	1
1	2	1	0	1	0	1	1	1	0	1	0	1	1	1	0	1	0	1	1
1	1	1	0	1	0	1	1	1	0	1	0	1	1	1	0	1	0	1	1
1	0	0	$\bar{1}$	0	$\bar{1}$	0	0	1	0	1	0	1	1	1	0	1	0	1	1
1	$\bar{1}$	0	$\bar{1}$	0	$\bar{1}$	0	0	0	$\bar{1}$	0	$\bar{1}$	0	0	0	$\bar{1}$	0	$\bar{1}$	0	0
1	$\bar{2}$	0	$\bar{1}$	0	$\bar{1}$	0	0	0	$\bar{1}$	0	$\bar{1}$	0	0	0	$\bar{1}$	0	$\bar{1}$	0	0
0	2	0	1	0	1	0	0	1	0	1	0	1	1	0	$\bar{1}$	0	1	0	0
0	1	0	1	0	1	0	0	0	$\bar{1}$	0	$\bar{1}$	0	0	0	$\bar{1}$	0	1	0	0
0	0	0	1	0	1	0	0	0	$\bar{1}$	0	$\bar{1}$	0	0	0	$\bar{1}$	0	1	0	0
0	$\bar{1}$	0	1	0	1	0	0	0	$\bar{1}$	0	$\bar{1}$	0	0	0	$\bar{1}$	0	1	0	0
0	$\bar{2}$	$\bar{1}$	0	$\bar{1}$	0	$\bar{1}$	$\bar{1}$	0	$\bar{1}$	0	$\bar{1}$	0	0	0	$\bar{1}$	0	1	0	0
$\bar{1}$	2	0	1	0	1	0	0	0	1	0	1	0	0	0	1	0	1	0	0
$\bar{1}$	1	0	1	0	1	0	0	0	1	0	1	0	0	0	1	0	1	0	0
$\bar{1}$	0	$\bar{1}$	0	$\bar{1}$	0	$\bar{1}$	$\bar{1}$	0	1	0	1	0	0	$\bar{1}$	0	$\bar{1}$	0	$\bar{1}$	$\bar{1}$
$\bar{1}$	$\bar{1}$	$\bar{1}$	0	$\bar{1}$	0	$\bar{1}$	$\bar{1}$	$\bar{1}$	0	$\bar{1}$	0	$\bar{1}$	$\bar{1}$	$\bar{1}$	0	$\bar{1}$	0	$\bar{1}$	$\bar{1}$
$\bar{1}$	$\bar{2}$	$\bar{1}$	0	$\bar{1}$	0	$\bar{1}$	$\bar{1}$	$\bar{1}$	0	$\bar{1}$	0	$\bar{1}$	$\bar{1}$	$\bar{1}$	0	$\bar{1}$	0	$\bar{1}$	$\bar{1}$
$\bar{2}$	2	1	0	1	0	1	$\bar{1}$	0	1	0	1	0	0	$\bar{1}$	0	$\bar{1}$	0	$\bar{1}$	$\bar{1}$
$\bar{2}$	1	1	0	1	0	1	$\bar{1}$	$\bar{1}$	0	$\bar{1}$	0	$\bar{1}$	$\bar{1}$	$\bar{1}$	0	$\bar{1}$	0	$\bar{1}$	$\bar{1}$
$\bar{2}$	0	1	0	1	0	1	$\bar{1}$	$\bar{1}$	0	$\bar{1}$	0	$\bar{1}$	$\bar{1}$	$\bar{1}$	0	$\bar{1}$	0	$\bar{1}$	$\bar{1}$
$\bar{2}$	$\bar{1}$	1	0	1	0	1	$\bar{1}$	$\bar{1}$	0	$\bar{1}$	0	$\bar{1}$	$\bar{1}$	$\bar{1}$	0	$\bar{1}$	0	$\bar{1}$	$\bar{1}$
$\bar{2}$	$\bar{2}$	0	$\bar{1}$	0	$\bar{1}$	0	$\bar{2}$	$\bar{1}$	0	$\bar{1}$	0	$\bar{1}$	$\bar{1}$	$\bar{1}$	0	$\bar{1}$	0	$\bar{1}$	$\bar{1}$

The choices in Equation (3.2.13) are given in Table 3 for all values of σ'_i and for all three adders. Table 4 gives the sum digit a'_i for all possible combinations of values of σ_i, σ_{i+1}, σ_{i+2} and for all three adders. It shows the overall result of Equations (3.2.11) through (3.2.14). Specific design details of radix two signed digit adders are discussed in papers by Robertson (16) and Borovec (7).

3.2.6 Addition Overflow Correction

The sum produced by the adder may have a non-zero a'_o digit. This is the indication of potential overflow; it is only potential overflow because it may be possible to recode this sum into an arithmetically equivalent representation A'' which has $a''_o = 0$ (e.g., $A' = 1.00\bar{1}\ldots$ may be recoded into $A'' = 0.111\ldots$).

The arithmetic unit may be designed to shift all such sums until $a''_o = 0$, or to attempt to recode the sums prior to shifting. It will be shown below that normalization is required to perform division, so the ability to perform the recodings mentioned above must exist in the arithmetic unit. Therefore the decision above will have a minor effect on the complexity of the arithmetic unit. If the arithmetic unit shifts all non-zero sum digits into the DPUs without attempting to recode them,

additions will be faster but will tend to discard digits of the result unnecessarily.

3.3 Multiplication Considerations

As indicated in Section 3.1, the requirements of division and normalization dictate that the operands are placed in the DPUs so that the digits of each of the operands are available in order of decreasing significance. Hence, the multiplication algorithm must be right-directed; i.e., the greater the significance of a given multiplier digit, the earlier it determines what multiple of the multiplicand will be added to the partial product. This, in turn, implies that the partial product must be shifted left one digital position with respect to the multiplicand for each step in the algorithm.

One possibility is that of shifting the multiplicand one digital position to the right each step. While this method has merit and may be sufficient in some circumstances, it does not allow the product to be determined to the greatest precision possible, as the digits shifted beyond the structure containing addition capability are functionally truncated.*

---

*Multiple precision arithmetic operations, while possible in the limited connection arithmetic unit, appear to require an excessive amount of time to perform. Hence, a single precision product may be all that is required. This method would then be adequate.

We will go on to discuss an implementation of multiplication in the limited connection arithmetic unit which allows the product to be determined to maximum precision. It is clear that the partial product, not the multiplicand, must be shifted during each step, since all digits of the multiplicand must take part in each step of the multiplication.

3.3.1 Manipulation of the Partial Product

Since the arithmetic unit possesses a single length adder, a scheme similar to that employed on the Illiac (14) and the IBM 7094 (11) must be incorporated in order to retain all the product digits determined. The digits of the multiplier which are retired can be used to make room for the digits of the product which need no longer be processed by the adder (although this destroys the multiplier in the process).

There are two problems associated with the adaptation of this technique to the proposed arithmetic structure. The first problem is the necessity of communicating information from the most significant digital position of the adder unit to the least significant digital position of the multiplier register. Not only are the DPUs containing these digital positions of the operands not in direct communication with one another, they are not performing the same micro-instruction, as can be seen

from the example of Section 2.2. Special connections causing them to exchange this information would, therefore, be of no use. The solution to this problem is to send the value of the digit to be placed in the least significant digital position with the left shift multiplier micro-instruction. In this way, the product digit which is to be placed in the least significant position of the multiplier register is available to the least significant DPU when that DPU is executing the shift. The interface between the DPU containing the most significant digit of the operands and the Primitive Control Unit should also be identical to that between any two adjacent DPUs. The Primitive Control Unit should contain a "multiplier register" into which the information to be shifted into the least significant digital position is placed and which is caused to exchange multiplier digits with the most significant DPU. This not only allows the DPU containing the most significant digits of the operands to be treated exactly like any other DPU, but affords a method of making multiplier digits available to the primitive control unit. The digit which is in this register at the completion of the exchange between the register and the multiplier register of the first DPU is the next multiplier digit to be employed.*

---

*This is true if one or more exchanges, dependent on whether a recoder is incorporated into the design, precede the first use of the adder.

3.3.2 A Multiplication Example

An example of the procedure just described is shown in Figure 8. In this example the following assumptions are made:

1) the operands are assumed to be three digits in length,
2) three registers (M, A, and Ø) take part in the multiplication. These registers play the following roles in the process. The A register contains zero prior to the multiplication and contains the least significant digits of the product at the conclusion of the process. The M register initially contains the multiplier and holds the most significant digits of the product at the conclusion of the multiplication. The Ø register contains the multiplicand throughout the process;
3) $\alpha = 0$ for all micro-instructions*,
4) the adder never causes overflow (i.e., $a_o = 0$ following an addition),
5) multiplier recoding is not employed,
6) the 'Pr Data' register is the only data register in the

*This assumption is definitely unrealistic. The figure is made more compact by this assumption.

PCU which takes part in the multiplication. Its contents define the multiple of the Ø register to be added to the A register when an addition (+) micro-instruction is performed. Just prior to the execution of a left shift micro-instruction by DPU_1 the 'Pr Data' register contains the digit which is to be placed in the least significant DPU (DPU_3) by that micro-instruction. The 'Pr Data' register receives the digit shifted out of DPU_1 when DPU_1 executes a left shift micro-instruction.

The following conventions are used in Figure 8:

M_j is the heading of the column indicating the contents of the M register of DPU_j,

A_j is the heading of the column indicating the contents of the A register of DPU_j,

μ_j is the heading of the column indicating the micro-instruction which has just been executed by DPU_j,

'Pr Data' is the heading of the column indicating the contents of the register in the PCU taking part in the multiplication,

Line No.	M_1	M_2	M_3	Pr. Data	A_1	A_2	A_3	μ_1	μ_2	μ_3
1	m_1	m_2	m_3	x	0	0	0	-	-	-
2	m_2			m_1				EM	-	-
3		m_3		0	${}_1P_1$			$+m_1$	EM	-
4			x	P_1	${}_1P_2$	${}_1P_2$		EA	$+m_1$	EM
5	m_3			m_2		${}_1P_3$	${}_1P_3$	EM	EA	$+m_1$
6		x		0	${}_2P_1$		0	$+m_2$	EM	EA
7			P_1	P_2	${}_2P_2$	${}_2P_2$		EA	$+m_2$	EM
8	x			m_3		${}_2P_3$	${}_2P_3$	EM	EA	$+m_2$
9		P_1		0	${}_3P_1$		0	$+m_3$	EM	EA
10			P_2	P_3	${}_3P_2$	${}_3P_2$		EA	$+m_3$	EM
11	P_1			x		${}_3P_3$	${}_3P_3$	EM	EA	$+m_3$
12		P_2					0	-	EM	EA
13	P_1	P_2	P_3	x	P_4	P_5	0	-	-	EM

Figure 8. Multiplication example

+k indicates that the corresponding DPU is performing an addition micro-instruction in which

$$A \leftarrow A + k \cdot \emptyset,$$

EM or EA indicates that the corresponding DPU is performing a left shift of the M or A register respectively,

m_i is the i^{th} multiplier digit,

${}_jp_i$ is the i^{th} digit of the portion of the j^{th} partial product which is in the A register, and

P_j is the j^{th} digit of the product and is

$$P_j = {}_jp_1 \qquad 1 \leq j \leq 3$$

$$P_j = {}_3p_{j-2} \qquad j = 4, 5$$

$$P_6 = 0.$$

The arithmetic unit is prepared for the multiplication at time 1. The M register contains the multiplier (m_1 m_2 m_3), the $\emptyset$ register* contains the multiplicand, and the A register contains zero. At time 2 a 'left shift M register' micro-instruction (EM) is launched. This causes the most significant multiplier digit (m_1) to be placed in 'Pr Data'. An addition micro-instruction is launched at time 3, which causes the first partial product (${}_1p_1$ ${}_1p_2$ ${}_1p_3$) to be formed in the A register and clears 'Pr Data'.

---

*The $\emptyset$ register is not shown in the figure because it remains unchanged throughout the multiplication.

A 'left shift A register' micro-instruction (EA) is launched next. This causes the most significant digit of the partial product ($_1p_1$) to be placed into 'Pr Data', which is also the most significant digit of the product, P_1. A 'left shift M register' micro-instruction is issued at time 5. This causes:

a) P_1 to be carried along with the micro-instruction and placed into DPU_3,

b) m_2 to be shifted into 'Pr Data', and

c) causes the remainder of the M register to be shifted left one DPU (or digital position).

The second of the repetitive steps now begins at time 6 with the launching of an addition micro-instruction. When all five of the repetitive steps is completed by all DPUs (time 21)

a) the most significant part of the product (P_1 P_2 P_3) is contained in the M register,

b) the least significant part of the product (P_4 P_5 0) is contained in the A register, and

c) 'Pr Data' contains the digit (x) that it contained just prior to the multiplication.

3.3.3 The Multiplication Overflow Recoder

A second problem associated with the shifting of the partial product in a right multiplier is that not one but two digits are shifted out at each step. That is, in addition to the most significant digit being shifted out, a non-zero overflow digit may have been formed and must be taken into account. It is clear that the digits shifted out can be converted so that only one digit need be stored in the multiplier register during each step, since the product of two numbers which are less than one in magnitude is also less than one in magnitude.

The multiplication algorithm can be characterized by

$$P_{-1} = 0 \tag{3.3.1}$$

$$P_i = r \cdot P_{i-1} + m_i \cdot \emptyset \quad i=0,1,\ldots,n \tag{3.3.2}$$

where

P_i is the i^{th} partial product,

m_i is the i^{th} digit of the multiplier,

r is the radix, and

$\emptyset$ is the multiplicand.

The product is then

$$M \cdot \emptyset = r^{-n} P_n \qquad (3.3.3)$$

where

M is the multiplier, i.e., $M = \sum_{i=0}^{n} m_i r^{-i}$.

Each partial product can be written in the form

$$P_i = \sum_{j=h}^{i} {}_ip_j r^j + z_i + \sum_{j=2}^{n} {}_ia_j r^{-j} \qquad (3.3.4)$$

where

${}_ip_j$ is the digit of the i^{th} partial product which has weight r^j, $|{}_ip_j| \leq (r-1)$,

h is an integer to be determined,

z_i is the i^{th} value of a variable indicating the value of the overflow from the adder which has not been included in the ${}_ip_j$'s.

${}_ia_j$ is the j^{th} digit of the i^{th} partial product, which will remain in the adder when the $i+1^{st}$ step of the algorithm is performed.

Applying the representation of P_i above to Equation (3.3.2) we obtain

$$P_{i+1} = \sum_{j=h}^{i} {}_ip_j r^{j+1} + rz_i + \sum_{j=2}^{n} {}_ia_j r^{1-j} + \sum_{j=1}^{n} m_{i+1} \phi_j r^{-j} \qquad (3.3.5)$$

Recognizing that the adder combines the last two portions of the partial product, i.e.

$$\sum_{j=0}^{n} {}_{i+1}a_j r^{-j} = \sum_{j=2}^{n} {}_{i}a_j r^{1-j} + \sum_{j=1}^{n} m_{i+1} \phi_j r^{-j} \tag{3.3.6}$$

we get

$$P_{i+1} = \sum_{j=h}^{i} {}_{i}p_j r^{j+1} + rz_i + {}_{i+1}a_0 + {}_{i+1}a_1 r^{-1} + \sum_{j=2}^{n} {}_{i+1}a_j r^{-j} \tag{3.3.7}$$

Identifying

$${}_{i+1}p_{j+1} = {}_{i}p_j, \tag{3.3.8}$$

yields

$$P_{i+1} = \sum_{j=h+1}^{i+1} {}_{i+1}p_j r^{j} + rz_i + {}_{i+1}a_0 + {}_{i+1}a_1 r^{-1} + \sum_{j=2}^{n} {}_{i+1}a_j r^{-j} \tag{3.3.9}$$

To obtain the form of Equation (3.3.2), the following relation must hold

$${}_{i+1}p_h r^{h} + z_{i+1} = rz_i + {}_{i+1}a_0 + {}_{i+1}a_1 r^{-1} \tag{3.3.10}$$

Hence the problem of converting the digits shifted out of the adder during a multiplication into a sequence of single digits is the determination of the value of h and the range of values taken on by the z_i in Equation (3.3.10) above.

In order to simplify further discussion, we will let

$$\gamma_i = {}_i p_h \tag{3.3.11}$$

$$\beta_i = r\, {}_i a_0 + {}_i a_1, \tag{3.3.12}$$

and

$$Z_i = r\, z_i. \tag{3.3.13}$$

Multiplying Equation (3.3.10) by r and making the above substitutions, it becomes

$$\gamma_{i+1}\, r^{h+1} + Z_{i+1} = r\, Z_i + \beta_i. \tag{3.3.14}$$

To determine h and the set of values taken on by Z_i in the equation above, we will use the analysis formulated by Rohatsch (21).

We see from Equation (3.3.12), the characteristics of signed digit adders, and the assumptions of symmetric number systems that

$$\beta_i \in B_{(r-\ell)\,(r+1)}. \tag{3.3.15}$$

Clearly

$$S(\beta_i) > r, \tag{3.3.16}$$

so the set of values taken on by $rZ_i + \beta_i$ are therefore contiguous in the terminology of Rohatsch.

Hence, the set of values taken on by $\gamma_i r^{h+1} + Z_{i+1}$ must also be contiguous. This implies that

$$S(Z_{i+1}) \geq r^{h+1} . \qquad (3.3.17)$$

Since β_i and γ_{i+1} are symmetric, we will assume that Z_i and Z_{i+1}, are also symmetric. Therefore, we will assume that

$$Z_i \leq \frac{r^{h+1} - \delta}{2} \qquad (3.3.18)$$

where $\delta = r \text{ Mod } 2$.

The set defined by the left side of Equation (3.3.14) must contain the set defined by the right side, and the largest element of the former must be larger than the largest element of the latter*, or

$$r^{h+1} \cdot (r-\ell) + \frac{r^{h+1} - \delta}{2} \geq r \cdot \left(\frac{r^{h+1} - \delta}{2}\right) + (r-\ell) \cdot (r+1) \qquad (3.3.19)$$

After manipulation to isolate ℓ, this becomes

$$\ell \leq \frac{r}{2} + \frac{1}{2} - \frac{(r-1)(r+1-\delta)}{2(r^{h+1} - r - 1)} \qquad (3.3.20)$$

In this form it is not difficult to see that h = 1 may be employed for all but the minimally redundant ($\ell = \frac{r}{2}$) even radix

---

*By symmetry, the smallest element of the former will then be smaller than the smallest element of the latter.

number systems. Therefore, the overflow digit recoder requires approximately r^3 states to retain the value of z_i when the minimum redundancy even radix representations are employed. In all other cases approximately r^2 states will suffice.

For the radix 2 arithmetic unit, the $h = 2$ overflow digit recoder must be employed as $r = 2$, $\ell = 1$ must be considered a minimally redundant system.

3.4 Multiplier Recoding

The expected time to perform a multiplication can be decreased in this arithmetic unit, as in all arithmetic units employing the shift and add algorithm to implement multiplication, if the fractional part of the multiplier is recoded to optimize the operations to be performed.

The multiplier recoding employed in this arithmetic unit should maximize the number of zero digits in the multiplier. A step of the multiplication algorithm for which the controlling multiplier digit is non-zero consists of two substeps, the shift and the addition; while a step for which the multiplier digit is zero consists only of the shift. A shift can be launched when only two modules have completed the previous micro-instruction, and an addition requires three

modules to have completed the previous micro-instruction. Hence, the addition cannot be launched until the last micro-instruction associated with the previous step of the algorithm has been executed by the first four modules. The latter step, therefore, takes at least twice as long to perform as the former, and hence the goal of maximizing the number of zeros in the multiplier.

3.4.1 Recoder Development Philosophy

We will now look at the problem of devising a multiplier recoder for a radix 2 arithmetic unit which employs Adder 2 (see Section 3.2).

This adder is such that the probability of digits with value zero in the sum is 1/2, as shown in Appendix I. It is possible to increase this statistic to 2/3 (15), making possible a potential decrease of 1/9 in the time required to perform the repetitive steps of the multiplication. Since the time required to do the other operations necessary to perform a multiplication is expected to be a very small fraction of the time required to do the repetitive steps, the multiplication time is expected to be decreased by this same amount, 1/9, by the inclusion of a recoder.

The work on such recoders with which the author is familiar are only applicable to numbers in non-redundant form. An approach for extending this work to radix two signed-digit numbers is discussed below. The recoder is assumed to consist of two processors acting in cascade. The first of these is an assimilator which reduces the redundancy of the number as much as possible. The second is a recoder which converts the assimilated intermediate representation of the number to one in which the probability of zero is maximized.

If it is possible to show that the recoder produces the same number of zeros for all arithmetically equivalent intermediate representations, then the probability of zero digits is related only to the distribution of values presented to the recoder and not to the distribution of their specific representations. We will define an assimilator which accepts signed-digit numbers and converts them into numbers which are in conventional form except for the possible replacement of $10\cdots00$ sequences by $01\cdots12$ sequences. We will then show that one of the minimal right directed recoders developed by Penhollow (15) can be extended to produce the same number of zero digits for all possible occurrences of the two arithmetically equivalent sequences above. Therefore this assimilator - recoder

cascade will recode radix 2 signed-digit numbers into minimal form.

3.4.2 The Assimilator

The proposed assimilator is shown in Table 5. The terminology employed in all of the tables of this section is as follows: The input representation is of the form

$$X = \sum_{i=1}^{n} x_i 2^{-i}, \tag{3.4.1}$$

that of the intermediate representation is

$$Y = -y_0 + \sum_{i=1}^{n} y_i 2^{-i} \tag{3.4.2}$$

and the recoded representation is

$$Z = \sum_{i=0}^{n} z_i 2^{-i} \tag{3.4.3}$$

where

x_i and z_i are signed digits; $x_i, z_i \in \{1, 0, \bar{1}\}$

y_i is a digit of the assimilated intermediate representation, $y_i \in \{0, 1, 2,\}$

$\bar{1}$ indicates the -1 digit.

\- indicates the entry does not affect the choice.

$\dot{1}$ indicates any number of consecutive one digits (including none).

Table 5. Right-Directed Assimilator.

Choose		Known			
b_i	y_i	b_{i-1}	x_i	x_{i+1}	x_{i+2}
$\bar{1}$	0	$\bar{1}$	$\bar{1}$	$\bar{1}$	-
$\bar{1}$	0	$\bar{1}$	$\bar{1}$	0	$\bar{1}$
$\bar{1}$	0	$\bar{1}$	$\bar{1}$	0	0
0	1	$\bar{1}$	$\bar{1}$	0	1
0	1	$\bar{1}$	$\bar{1}$	1	-
$\bar{1}$	1	$\bar{1}$	0	$\bar{1}$	-
$\bar{1}$	1	$\bar{1}$	0	0	$\bar{1}$
$\bar{1}$	1	$\bar{1}$	0	0	0
0	2	$\bar{1}$	0	0	1
0	0	0	0	-	-
$\bar{1}$	0	0	1	$\bar{1}$	-
$\bar{1}$	0	0	1	0	$\bar{1}$
$\bar{1}$	0	0	1	0	0
0	1	0	1	0	1
0	1	0	1	1	-

Table 6. Simplest Right-Directed Extended Recoder.

Choose		Known			
c_i	z_i	c_{i-1}	y_i	y_{i+1}	y_{i+2}
0	0	0	0	0	-
0	0	0	0	1	0
1	1	0	0	1	1
1	1	0	0	1	2
1	1	0	0	2	-
0	1	0	1	-	-
1	$\bar{1}$	1	0	-	-
0	$\bar{1}$	1	1	0	0
1	0	1	1	0	1
1	0	1	1	0	2
1	0	1	1	1	-
1	0	1	1	2	-
0	0	1	2	-	-

Table 7. Cases in Which a 2 Digit Is Presented to the Recoder Above.

Mode Digit	Digit Sequences Containing 2 Digits						Arithmetically Equal Digit Sequences						Recoder Output
0	0	0	0	$\dot{1}$	2	0	0	0	1	$\dot{0}$	0	0	=
0	0	1	0	$\dot{1}$	2	0	0	1	1	$\dot{0}$	0	0	≠
0	1	0	0	$\dot{1}$	2	0	1	0	1	$\dot{0}$	0	0	=
1	0	1	0	$\dot{1}$	2	0	0	1	1	$\dot{0}$	0	0	=
1	1	0	0	$\dot{1}$	2	0	1	0	1	$\dot{0}$	0	0	≠
1	1	1	0	$\dot{1}$	2	0	1	1	1	$\dot{0}$	0	0	=

The basic relationship of the assimilator is

$$y_i = x_i + b_i - 2b_{i-1} \qquad 1 \leq i \leq n \qquad (3.4.4)$$

where b_i is a borrow digit, $b_i \varepsilon \{0, \bar{1}\}$.

x_i and b_{i-1} are known, y_i and b_i are to be chosen.

3.4.3 The Extended Penhollow Minimal Recoder

Table 6 is an extension of the Penhollow simplest minimal right-directed recoder (15). The equation governing the recoder is

$$z_i = y_i + c_i - 2c_{i-1} \qquad (3.4.5)$$

where

c_i is a carry digit, $c_i \varepsilon \{0, 1\}$.

y_i and c_{i-1} are known, z_i and c_i are to be chosen.

This extended recoder accepts the output of the assimilator described above. The output of the recoder was determined for the six cases (see Table 7) in which the recoder encounters a 2 digit. The output was compared with the output which would have been produced if the assimilator had been able to produce an intermediate representation which did not contain the 2 digit. In all six cases, the recoder produced an equal number of 0 digits whether the assimilator output contained a 2 digit or not. The output of the assimilator extended recoder therefore has the same shift average as the Penhollow recoder(i.e., 3).

In four of the six cases, the output of the recoder is the same whether the assimilator produced a 2 digit or not. In both of the other cases the recoder output had more 1 digits and fewer $\bar{1}$ digits when the assimilated number contained a 2 than when it didn't. Hence, the probability of 1 and $\bar{1}$ digits in the recoder output are unequal and not independent of the digit sequence used to represent the number. For example, both '0 1 1 0 0 1' and '0 1 1 1 $\bar{1}$ $\bar{1}$' represent 25_{10}, yet the first is recoded '0 1 1 0 0 1' while the second is recoded '1 0 $\bar{1}$ 0 0 1'. Appendix II presents the overall recoder table and some additional discussion.

3.5 Normalization Considerations

There are several situations in which it is necessary to normalize numbers, that is, restrict the range of values which the fractional part may assume. These are the preparation of operands and the processing of results.

3.5.1 Definition of Normalized Numbers and Their Range of Values

The major design consideration in the implementation of normalization is the choice of data to examine to determine if additional processing of the number is necessary.

We will define a signed digit to be normalized when either:

1. $|x_1| \geq 2$,

2. $|x_1| = 1$, $x_2 = x_3 = \ldots = x_{\tau-1} = 0$ and $x_\tau \cdot x_1 \geq 1$, $\tau \leq \nu$,

3. $|x_1| = 1$, and $x_i = 0$, $i = 2, 3, \ldots, \nu$, or

4. $|x_1| = 1$, and $x_1 \cdot x_2 \geq (1-\ell)$ if $\ell > 1$.

where x_i is the i^{th} digit of the number, with weight

$$r^{-i} \text{ (ie, } X = \sum_{i=1}^{n} x_i r^{-i}\text{)}, \quad x_i \varepsilon B_{r-\ell},$$

ν is the number of digits examined to determine if X is normalized.

ℓ is a integer parameter indicating the redundancy of the number system, $1 \leq \ell \leq \frac{r}{2}$.

X therefore consists of two components X_i and $X_{\overline{i}}$, and

$$X = X_i + X_{\overline{i}},$$

where

X_i is the value of the first i digits of the number,

$$\text{i.e., } X_i = \sum_{j=1}^{i} x_j r^{-j}, \text{ and}$$

$X_{\overline{i}}$ is the value of the remaining digits of the number,

$$\text{i.e.,}\quad X_{\overline{i}} = \sum_{j=i+1}^{n} x_j r^{-j}.$$

From the definition of normalized numbers above, we see that, for $\ell = 1$

$$r^{-1} \leq |X_\nu| \leq (1 - r^{-\nu}), \text{ and}$$

$$0 \leq |X_{\overline{\nu}}| \leq (r^{-\nu} - r^{-n}).$$

These can be combined to yield

$$r^{-1} - r^{-\nu} + r^{-n} \leq |X| \leq 1 - r^{-n} \qquad (3.5.1)$$

When $\ell > 1$, we see from condition 4 that $\nu = 2$ and the analysis becomes

$$\frac{r+1-\ell}{r^2} \leq |X_2| \leq (r-\ell)\frac{(r+1)}{r^2},$$

$$0 \leq |X_{\overline{2}}| \leq \frac{(r-\ell)}{(r-1)} \cdot \left[\frac{1}{r^2} - \frac{1}{r^n}\right],$$

which yields

$$\frac{r+1-\ell}{r^2} + \frac{(\ell - r)}{(r-1)} \cdot \left[\frac{1}{r^2} - \frac{1}{r^n}\right] \leq |X| \leq \frac{(r-\ell)}{(r-1)} \cdot \left(1 - \frac{1}{r^n}\right) \qquad (3.5.2)$$

We may see from the above that the range to which numbers can be normalized is fixed for all but the maximally redundant numbers. For maximally redundant numbers, the range of values decreases as the number of digits increases.

3.5.2 Normalization Recodings

The procedure for normalizing signed-digit numbers is complicated by the existence of representations for which x_1 is not zero but which do not meet the definition of normalized numbers. These representations must be recoded into an arithmetically equivalent representation which, after shifting out leading 0 digits, does meet the definition.

The recoding converts the most significant digits of this representation,

$$x_1 = \pm 1,\ x_2 = x_3 = \ldots = x_{\tau-1} = \overline{+}(\ell-1),\ x_\tau = \overline{+}(r-a) \qquad (3.5.3)$$

into the following digits

$$x_1 = 0,\ x_2 = x_3 = \ldots = x_{\tau-1} = \pm(r-\ell),\ x_\tau = \pm a \qquad (3.5.4)$$

where either the upper or the lower sign is chosen uniformly in the description, and τ is the number of digits altered by the recoding, $\tau \geq 2$.

A decimal example of this procedure, assuming $\ell = 2$, is the recoding of ' $\overline{1}$ 1 1 6 5 $\overline{3}$ ' into ' 0 $\overline{8}$ $\overline{8}$ $\overline{4}$ 5 $\overline{3}$ '. In this example $x_1 = -1$, $x_2 = x_3 = +(2-1)$, and $x_4 = +6$.

3.5.3 Methods of Normalization

There are two basic methods of normalizing numbers in this arithmetic unit. They differ in the manner in which they obtain information about the number.

In the first method, the values of the most significant digits are sensed only by shifting them into the PCU. This can be thought of as over-normalization followed by restoration, since the number will have a non-zero integer part stored in the PCU just prior to restoration.

In the second method the values of required digits are transmitted to the PCU by some micro-instruction.

Provision for sensing the contents of the A register will be included to sense partial remainders during division. Hence, this normalization technique would not appreciably add to the complexity of the DPUs. In addition, this latter method affords the designer two options to minimize the time to perform the partial normalization.

The first option is the inclusion of sense micro-instructions detectors among the DPUs. Each time the primitive control unit receives notification that additional digits of the number are available it can determine whether it has sufficient information to complete the partial normalization and also whether additional

left shift micro-instructions must be issued. Note that the only case for which neither is true is the case for which the recoding may have to be performed. Hence, in all but this one case the time required to sense the number is largely overlapped with the time to perform the shifts required, and the time to partially normalize may be less than the time for a micro-instruction to propagate through ν digit processing units.

The second option is the inclusion of micro-instructions to perform the recodings indicated above. These micro-instructions would be such that they would add $\pm (r-1)$ to the register of the DPU containing the representation being normalized if it initially contained $\mp (\ell-1)$ and would then transmit the micro-instruction to its right neighbor. When the register of the DPU executing the micro-instruction contains any other digit, the micro-instruction causes $\pm r$ to be added to the register contents, hence affecting the $\mp (r-a)$ to $\pm a$ transformation. This is the form of the transformation required of the last digit to be altered. This last DPU does not pass the micro-instruction to its neighbor.

Comparisons between the two basic methods are very difficult to make because of the options available with the latter method.

3.5.4 Analysis of Radix Two Normalization Methods

Now we will look in detail at the problem of normalizing radix two numbers when $\nu = 2$. The choice of examining first two digits of the number is compatible with the implementation of the simplest division algorithm. Because of the small value of ν, we will assume that the arithmetic unit does not contain sense micro-instruction detectors and that the DPUs do not have micro-instructions to do the recodings.

Numbers in this system which do not require recoding during normalization are of the form

$$\underbrace{0 \ldots 0}_{i} \; x \; X \qquad\qquad i \geq 0$$

where x is either +1 or -1 for any particular number, and X is either 0 or x.

Numbers in this system which do require recoding are of the form

$$\underbrace{0 \ldots 0}_{i} \; x \; \bar{x} \underbrace{\bar{x} \ldots \bar{x}}_{j} \; X \qquad\qquad i \geq 0, \; j \geq 0$$

where $\bar{x} = -x$.

The above numbers will be referred to as (i) and (i, j) in the remainder of the discussion.

We will assume that the probability distribution of a given digital position is a function only of the value of the digit to its left, and that these probabilities* are

$$P(\text{leading zero}) = \frac{1}{2} \qquad P(0|1) = P(0|\bar{1}) = \frac{1}{2}$$

$$P(0|0) = \frac{1}{2} \qquad P(1|\bar{1}) = P(\bar{1}|1) = \frac{1}{3}$$

$$P(1|0) + P(\bar{1}\ 0) = \frac{1}{2} \qquad P(1|1) = P(\bar{1}|1) = \frac{1}{6},$$

where $P(y|z) = P(x_{i+1} = y, \text{ given } x_i = z)$.

Then the probabilities of the two numbers are

$$P((i)) = \frac{1}{3 \cdot 2^i}, \text{ and} \qquad (3.5.5)$$

$$P((i, j)) = \frac{5}{36 \cdot 2^i \cdot 6^j} \qquad (3.5.6)$$

Four methods for performing this normalization will be considered.

Method A consists of shifting the representation left until the portion of the representation shifted into the Primitive Control Unit can be recoded into two digits of the same sign or a non-zero digit followed by a zero. The two digits are then shifted back into the DPUs.

---

*This is based on the analysis of a radix two adder in Appendix I.

Method B consists of examining the digits contained in the two most significant DPUs. If the terminal digit is not detected, two left shifts are launched and the process repeated. If the terminal digit is detected, the additional shifts* necessary to normalize the number are performed and the process terminated.

Method C consists of examining the digits contained in the two most significant digital positions. If the terminal digit is not detected, one left shift is performed if the first non-zero digit is in the second digital position, two left shifts are performed otherwise. The process is then repeated. If the terminal digit is detected, then the appropriate final shifts, if any, are performed and the process terminated.

Method D consists of shifting the representation to be normalized left until a non-zero digit is shifted into the primitive control unit. The first two digital positions are then examined. If the terminal digit is not detected, two left shifts are performed. The (examination, two left shift) portion of the procedure is repeated until a terminal digit is detected and the appropriate final steps performed.

---

*The recoding is performed by shifting the digit to be changed into the primitive control unit, which recodes the digit and shifts the altered digit right (into DPU_1).

Table 8. Normalization Procedure for All Possible Radix 2 Signed Digit Numbers

Method	Class Number	Left Shifts	Number of Examinations	Right Shifts
A	(i)	i+2	0	2
	(i, j)	i+j+3	0	2
B	(i) i even	i	$\frac{i+2}{2}$	0
	(i) i odd	i+1	$\frac{i+3}{2}$	1
	(i, j) i+j even	i+j+2	$\frac{i+j+4}{2}$	1
	(i, j) i+j odd	i+j+3	$\frac{i+j+3}{2}$	1
C	(i) i even	i	$\frac{i+2}{2}$	0
	(i) i odd	i	$\frac{i+3}{2}$	0
	(i, j) i+j even	i+j+2	$\frac{i+j+4}{2}$	1
	(i,j) {i even, j odd}	i+j+2	$\frac{i+j+3}{2}$	1
	(i,j) {i odd, j even}	i+j+3	$\frac{i+j+5}{2}$	1
D	(i)	i+1	1	1
	(i, j) j even	i+j+3	$\frac{j+2}{2}$	1
	(i, j) j odd	i+j+2	$\frac{j+3}{2}$	1

Table 9. Average Number of Operations Required During a Normalization

Normalization Method	⟨L.S.⟩	⟨R.S.⟩	⟨E⟩
A	$3\frac{2}{5}$	2	0
B	$1\frac{43}{45}$	$\frac{5}{9}$	$1\frac{32}{35}$
C	$1\frac{11}{15}$	$\frac{1}{3}$	$2\frac{1}{105}$
D	$2\frac{2}{5}$	1	$1\frac{2}{35}$

Table 8 gives the number of left shifts, right shifts, and examinations which must be performed for each possible number and for each of the above methods of normalization.

Applying the probabilities given in (3.5.5) and (3.5.6) and assuming that the numbers are not limited in length*, one obtains the average number of left shifts, right shifts, and examinations shown in Table 9. Assuming that the time to perform a left shift is the same as the time to perform a right shift and neglecting the time required to make decisions, the expected number of shift times required to perform a normalization is

$$\langle T \rangle = \langle LS \rangle + \langle RS \rangle + \xi \cdot \langle E \rangle \qquad (3.5.7)$$

where

$\langle T \rangle$ is the expected number of shift micro-instruction times required to normalize an output of a symmetric base 2 signed digit adder.

$\langle LS \rangle$ is the expected number of left shift micro-instructions required.

---

*The error in these calculations is approximately 3×2^{-n}, and 2^{1-n} for the average number of left shifts and examinations required, respectively, where n is the number of digits in a number.

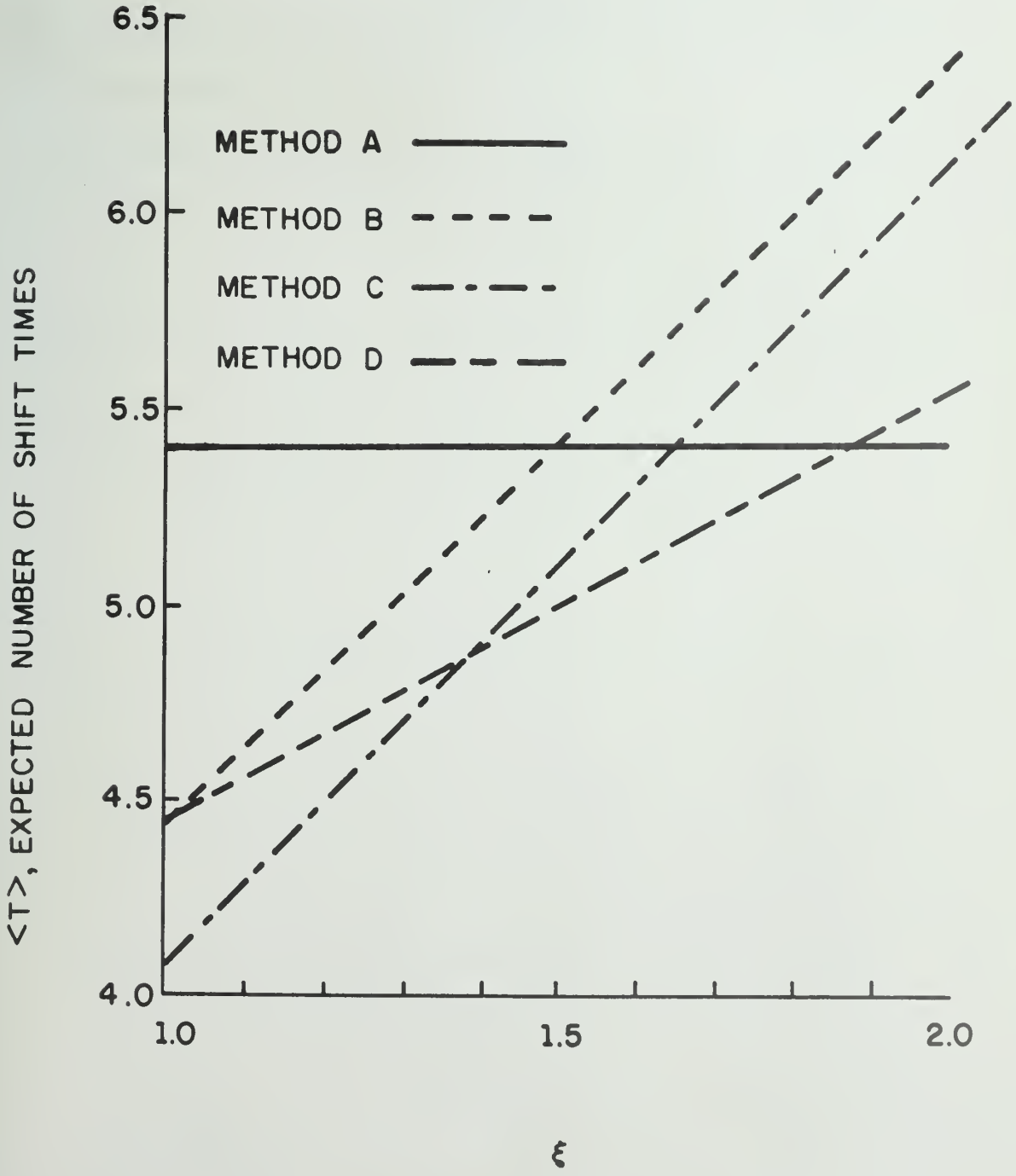


Figure 9. Average time to normalize a radix two signed digit number.

$\langle RS \rangle$ is the expected number of right shift micro-instructions required.

ξ is the ratio of the time required to transmit the two most significant digits of the number undergoing normalization to the primitive control unit to the time required to launch a shift micro-instruction.

$\langle E \rangle$ is the expected number of times the digits retained in the first two digital positions of the register containing the number being normalized must be transmitted to the primitive control unit.

A graph of $\langle T \rangle$ versus ξ as determined from Table 9 and (3.5.7) is presented as Figure 9. This graph shows that Normalization Method C is optimum when $\xi < 1\frac{2}{5}$, Method D when $1\frac{2}{5} < \xi < 1\frac{33}{37}$, and Method A when $\xi > 1\frac{33}{37}$. The value of ξ is approximately 2 if the 'red tape' is issuing micro-instructions is negligible and becomes smaller as the 'red tape' becomes more predominate. Hence, Method A is the optimum when 'red tape' is negligible; Methods D and then C become optimum as the 'red tape' increases.

3.6 Division Considerations

Division, like multiplication, must be performed by repetitive additions and shifts in the limited connection arithmetic unit because it contains a single adder. Unlike multiplication, however, the partial remainder must be examined periodically to determine one or more quotient digits, which will control the use of the adder in subsequent steps. That is, while the repetitive steps of division must be performed radix r because of the existence of a single adder, the quotient digit determination may be performed with radix r^{λ}, where λ is the number of radix r quotient digits determined in one comparison.

In the arithmetic unit under investigation, the time required to transmit a given number of digits of the partial remainder or divisor to the selection mechanism is directly proportional to the number of digits transmitted. The value of several digits of the partial remainder are required to determine one quotient digit; the value of one additional digit is required for each additional quotient digit. Hence, the total time spent obtaining information from the partial remainders decreases as the number of quotient digits determined in each step increases. This effect is opposed by two factors. The first is the accuracy to which the value of the divisor must be

known. The number of digits of the divisor whose value must be available to the quotient digit selection logic has the same form as that for the partial remainder; several digits for the first quotient digit, one for each additional quotient digit determined. The second factor is the time required for the quotient digit selection logic to yield the quotient digits after the appropriate information is presented to it. This last factor is not related in a simple way to the micro-instruction execution time. Therefore, rather than determining a specific division algorithm, the number of digits of the divisor and partial remainders which must be presented to the quotient digit selection logic will be determined as a function of the number of quotient digits selected per step, and the radix and redundancy of the numbers employed. The class of divisions requiring the minimum information of the value of the partial remainders and divisors will be analyzed.

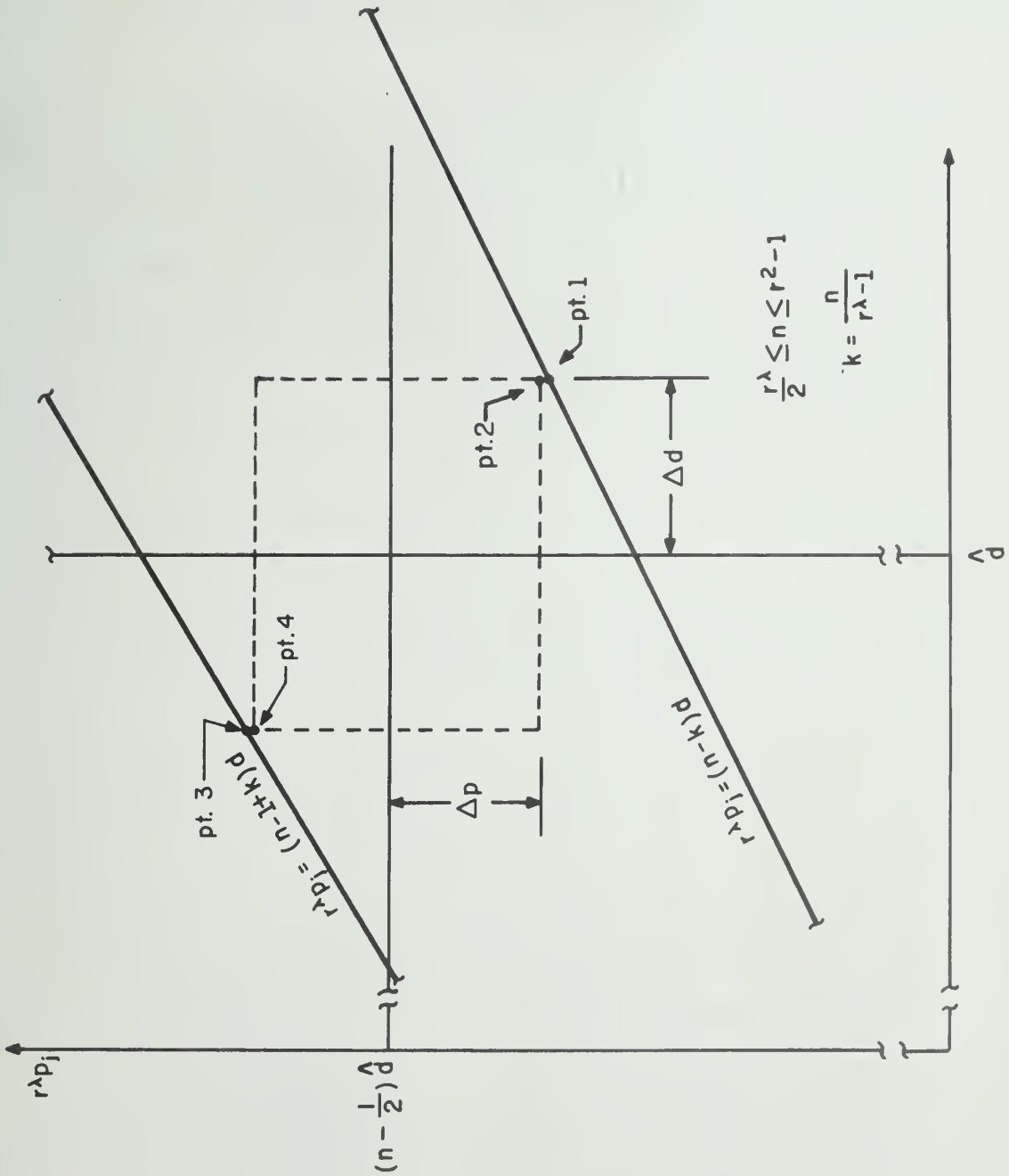


Figure 10. P-D plot for general SRT division.

3.6.1 Analysis of Division for Maximally Redundant Numbers

We will now begin the analysis of the number of digits of the partial remainder required to determine a number of quotient digits, extending the work of Robertson (18) and Atkins (3). The number of digits of the partial remainder and of the divisor which must be available to the quotient digit selection mechanism is determined by the situation depicted graphically in Figure 10. This figure is a P-D plot, as suggested by C. V. Frieman (18). The abscissa is the divisor value, the ordinate the value of the shifted partial remainder. The graph is divided into regions for which a given quotient digit value may be chosen.

The boundaries of such regions are lines of the form

$$r^{\lambda} p_j = (q-K)d \qquad (3.6.1)$$

and

$$r^{\lambda} p_j = (q+K)d \qquad (3.6.2)$$

where

r is the radix of the number representation system,

λ is the number of quotient digits determined by a single comparison (hence the division radix is effectively r^{λ}),

p_j is the value of the partial remainder at the conclusion of step j,

q is the quotient digit value which may be selected,

n is the largest possible quotient digit, i.e.,

$$q \in \{n, n-1, \ldots, 1, 0, \bar{1}, \ldots, \bar{n}\} ,$$

d is the divisor value,

K is a constant determined by the redundancy of the quotient and is

$$K = \frac{n}{r^{\lambda} - 1} . \qquad (3.6.3)$$

Note that since quotient digits will be determined by truncated versions of the divisor and partial remainder, it is only known that the point representing a given divisor and partial remainder is within a rectangular region of the truncated values. Hence, each such region must lie entirely within one of the quotient digit regions defined by Equations (3.6.1) and (3.6.2).

In Figure 10, line 1 represents the lower boundary of the region for which the choice q=n can be made. Line 2 is the upper boundary of the region for which q=n-1 may be chosen. The interior of the dashed-line rectangle represents the range of possible divisor, partial remainder pairs which have truncated values of $\hat{d}$ and $(n - \frac{1}{2})\hat{d}$, respectively where

$\hat{d}$ is the minimum positive truncated divisor value.

When this rectangle lies entirely within the two lines, all other such rectangles of the same size will be entirely within some quotient digit region*. Atkins has shown that insuring that point 3 on line 2 is not below point 4 on the rectangle guarantees that the rectangle is within the two lines. This condition is:

$$(n-1+K)\cdot(\hat{d} - \Delta d) \geq (n-\frac{1}{2})\hat{d} + \Delta p \qquad (3.6.4)$$

where

$\hat{d}$ is the smallest positive truncated divisor value,

Δp is the truncation error in the shifted partial remainder,

Δd is the truncation error in the divisor value.

Note that this equation is based on the assumption that the partial remainder will be shifted λ digital positions before the comparison is made. Since the most significant quotient digit has weight $r^{\lambda-1}$, the more usual procedure is to shift the partial remainder left only one digital position, and the λ next quotient digits are determined. The quotient digits are then disposed of beginning with the most significant digit.

---

*This is a conservative solution. It will be discussed later that solutions were found which had smaller values of ρ and δ.

One left shift is performed between each such pair of steps. Hence, the value of Δp determined by Equation (3.6.4) will be $r^{\lambda-1}$ larger than it will have to be if the method above is used for determining new partial remainders.

Taking the above consideration into account, Equation (3.6.4) can be manipulated into the following more useful form:

$$r^{\lambda-1} \cdot \Delta p + (n-1+K) \cdot \Delta d \leq (K - \frac{1}{2}) \cdot \hat{d} \qquad (3.6.5)$$

From the section on normalization we see that for maximally redundant numbers the division parameters are

$$\hat{d} = \frac{1}{r}, \quad \Delta d = r^{-\delta}, \quad \Delta p = r^{-\rho} \qquad (3.6.6)$$

where

δ is the number of digital positions of the divisor which are transmitted to the quotient digit selection device and

ρ is the number of digital positions to the right of the radix point which are transmitted to the quotient digit selection mechansim.

Table 10. Requirements for Performing Division with Maximally Redundant Numbers.

r	λ	ρ	δ
$r \geq 4$	$\lambda \geq 1$	$\lambda + 1$	$\lambda + 2$
$r = 3$	$\lambda \geq 2$	$\lambda + 2$	$\lambda + 2$
$r = 3$	$\lambda = 1$	2	3
$r = 2$	$\lambda \geq 3$	$\lambda + 2$	$\lambda + 3$
$r = 2$	$\lambda = 2$	4	4
$r = 2$	$\lambda = 1$	2	2

The maximum quotient 'digit' consists of λ digits all equal to (r-1), hence

$$n = \sum_{i=0}^{\lambda-1} (r-1)\, r^i = r^{\lambda} - 1 \qquad (3.6.7)$$

which yields from (3.6.3) that

$$K = 1 \ . \qquad (3.6.8)$$

Applying (3.6.6), (3.6.7) and (3.6.8) to (3.6.5),

$$r^{\lambda-1}\, r^{-\rho} + (r^{\lambda} - 1)\, r^{-\delta} \leq \frac{1}{2r} \qquad (3.6.9)$$

Now letting

$$\delta = \rho + 1 \qquad (3.6.10)$$

yields that

$$r^{\rho} \geq (4 - \frac{2}{r^{\lambda}})\, r^{\lambda} \qquad (3.6.11)$$

Therefore

$$\rho = \lambda + 1 \text{ and } \delta = \lambda + 2 \text{ for } r \geq 4. \qquad (3.6.12)$$

The other cases are, in general, no more difficult to solve. Optimum solutions are given in Table 10; the r=3, λ=1; r=2, λ=2 and λ=1 entries were not, however, determined from Equation (3.6.9). For these cases, requiring that the rectangle lie entirely within the region bounded by lines 1 and 2 in Figure 10 is overly conservative. In these cases it is

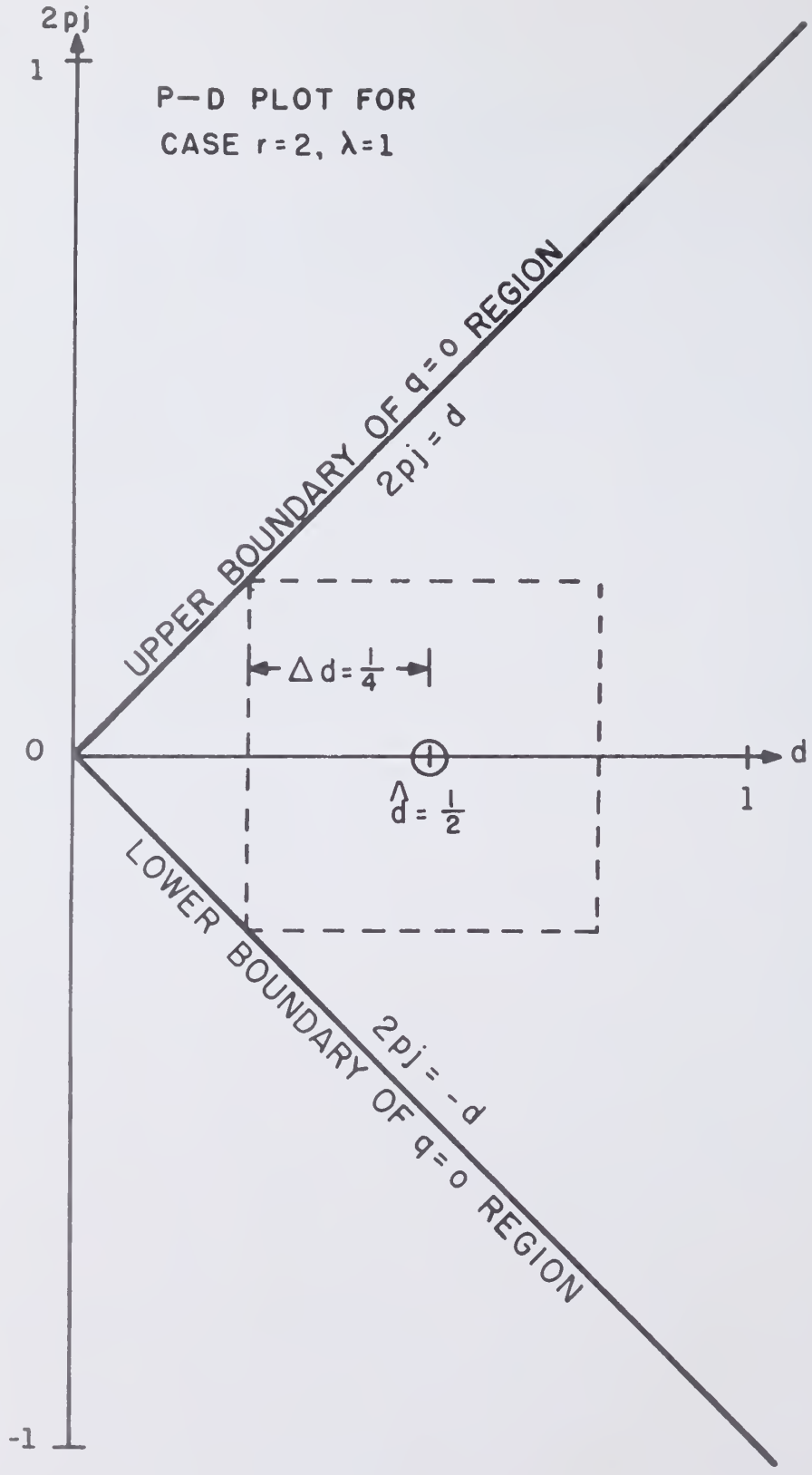


Figure 11. P-D plot used to determine δ for $r=2, \lambda=1$.

possible to choose ρ and δ such that point 2 of the rectangle coincides with point 1 of line 1. This condition is expressed algebraically as

$$r^{\lambda-1} r^{-\rho} + (r^{\lambda} - 2) r^{-\delta} \leq \frac{1}{2r} \qquad (3.6.13)$$

For all but r=2 and $\lambda = 1$, this equation determines ρ and δ. However, the coefficient of $(2)^{-\delta}$ is zero when r=2, $\lambda=1$ in Equation (3.6.13); hence δ could not be determined by that equation. To determine the value of δ, the P-D plot for this case was drawn. It was seen that Δ d must be such that the region of divisor values and partial remainder values centered at $(\frac{1}{2}, 0)$ will remain within the q=0 region. As seen in Figure 11, this value is $\Delta d = \frac{1}{4}$; hence $\delta=2$.

The reader should notice that when $\lambda = 1$, the values of only the first two digits of the partial remainder are required to select the quotient digit. Since, as we have seen in Section 3.2, the primitive control unit must receive information from the first two DPUs to detect overflow when addition or subtraction is performed, no additional data paths are required to sense the value of the partial remainder. For r=2, $\delta = 2$ also; indicating that no special data paths are necessary. For $r \geq 3$, $\delta = 3$; indicating that special provision would have to be made to transmit

the value of the divisor digit from the third DPU to the PCU. While it is possible to include a special data path from the third DPU to the PCU, an alternative which appears attractive is to normalize by method D as discussed in Section 3.4, since this allows the value of three digits of the operand to be accessible to the PCU. These are the digit shifted into the PCU, and the digits in the first two DPUs. Hence, it is possible to build an arithmetic unit for an arbitrary radix which has very regular and repetitive interconnections.

Whether this is adequate, or whether a division algorithm in which more than one quotient digit is determined at each comparison, can only be decided when some very implementation-dependent factors are taken into account. The major factors are the time required for the quotient digit selection logic to produce its output and the decrease in the speed of micro-instruction execution caused by the additional connection to certain of the data paths.

3.6.2 Analysis of Division for Other Number Systems

The analysis of division for an arithmetic unit in which other than a maximally redundant representation is employed is much more complicated. From the discussion of the preceeding

section, we see that, in general,

$$\hat{d} = \frac{r^2 - r - r\ell + 2\ell - 1}{r^2 (r-1)} + \frac{(r-\ell)}{(r-1)} r^{-\delta} \qquad (3.6.14)$$

$$\Delta p = \frac{(r-\ell)}{(r-1)} r^{-\rho} \qquad (3.6.15)$$

and

$$\Delta d = \frac{(r-\ell)}{(r-1)} r^{-\delta} \qquad (3.6.16)$$

The maximum quotient digit is

$$n = \sum_{i=0}^{\lambda-1} (r-\ell) r^{-i} = \frac{(r-\ell)}{(r-1)} (r^{\lambda} - 1) \qquad (3.6.17)$$

from which we obtain

$$K = \frac{r-\ell}{r-1} \qquad (3.6.18)$$

Applying the above to Equation (3.6.5) and doing some manipulation we obtain

$$r^{\lambda-1-\rho} + \left[\frac{(r-\ell)}{(r-1)} r^{\lambda} - \frac{3}{2} + \frac{(\ell-1)}{(r-1)}\right] r^{-\delta} \leq \frac{r-2\ell+1}{2r} \cdot \left[1 + \frac{(\ell-1)}{(r-\ell)(r-1)}\right]$$

Neglecting all but the first $r^{-\delta}$ and right-hand term and letting

$$\delta = \rho + 1,$$

and doing further manipulation we obtain

$$r^{\rho} \geq \frac{2r^{\lambda+1}(2r-\ell-1)}{(r-1)(r-2\ell+1)} . \qquad (3.6.19)$$

From which we find that

$$\rho = \lambda + 1 \text{ for } \ell \leq \frac{r}{2} - 1. \qquad (3.6.20)$$

When minimally redundant representations are employed in the arithmetic unit, the solution is

$$\rho = \lambda + 2 \text{ for } \frac{r}{2} - 1 < \ell \leq \frac{r}{2} \qquad (3.6.21)$$

The solutions given in lines (3.6.20) and (3.6.21) are pessimistic when λ is small because of the simplification which had to be made to obtain Equation (3.6.19). The analysis does show that, in general, the number of digits of the partial remainder and divisor which must be examined to determine the next quotient digits is a constant plus the number of quotient digits determined. The constant is a function of the radix and redundancy employed by the arithmetic unit.

3.6.3 Using the Multiplication Overflow Recoder During Division

The next problem associated with the implementation of division is the fact that the shifted partial remainder may become greater than unity, and hence could not be represented entirely in the accumulator register of the adder. Robertson (17) has shown that the shifted partial remainder satisfies*

$$\left| r\, p_j \right| \leq r \cdot \frac{(r-\ell)}{r-1} < r$$

The multiplication overflow recoder is shown in Section 3.3.3 to allow the value of the accumulator register to become as large as $\frac{r^2 - 1}{2}$ in absolute value. Hence, it can also be employed to act as an extension of the adder during division. Since in this arithmetic unit both multiplication and division are right-directed, there is little fundamental difference between the functions to be performed by the overflow recoder when it is employed for division and when it is employed for multiplication. During multiplication it will overflow and yield non-zero digits, while the choice of quotient digits prevents this during division. During division, the value retained must be

---

*Assuming that one shift, rather than λ, be made prior to determining the quotient digits. This is the same assumption made in obtaining Equation (3.6.5).

available to the quotient digit selection mechanism, while no such connection is necessary for multiplication. In every other respect, the two uses of the unit are identical. Hence, the multiplication overflow recoder can be employed to retain the integer part of the partial remainders by the addition of a very minimal amount of logic.

3.6.4 Placing the Quotient Digits into the DPUs

The final problem which the implementation of division poses is that of the storage of quotient digits and the accomodation of a double precision dividend. The most significant digit of the quotient is obtained first, hence shifting the register receiving the quotient right one position and inserting the newly determined quotient digit would result in the quotient being stored in reverse order. Hence, they must be "inserted" into the least significant digital position after the register is shifted left. This is essentially the same problem as accumulating product digits during multiplication. Just as in the case of multiplication, the solution is to use a left shift micro-instruction. In this case, the quotient digit to be stored is identified as the digit which is to be placed in the least significant digital position. The register that contains the least significant part of the dividend should be used to store the quotient digits (if it exists) since one digit of the least significant

part of the dividend must be moved to the accumulator for each quotient digit which must be stored. Furthermore, since the digits of the least significant part of the dividend must be sent to the accumulator in the order of decreasing significance, the digit shifted into the PCU by this left shift micro-instruction is the digit which must be moved to the accumulator. This digit can then be sent to the least significant digital position of the accumulator by identifying it as the digit which is to be placed there when the accumulator left shift micro-instruction is issued to multiply the partial remainder* by the radix.

3.6.5 A Division Example

An illustration of how division may be performed on a double-precision dividend is presented as Figure 12. In this example the Ø register contains the divisor. The A register initially contains the most significant part of the dividend and contains the remainder at the conclusion of the operation. The M register initially contains the digits of lesser significance and contain the quotient digits after the division.

---

*Which is contained in the accumulator during division.

Line No.	M_1	M_2	M_3	Pr. Data	A_1	A_2	A_3	μ_1	μ_2	μ_3
1	n_4	n_5	0	0	n_1	n_2	n_3	-	-	-
2				q_0				SA	-	-
3					$_1p_1$			$+\bar{q}_0$	SA	-
4	n_5			n_4		$_1p_2$		EM	$+\bar{q}_0$	SA
5		0		$_1p_1$	$_1p_2$		$_1p_3$	EA	EM	$+\bar{q}_0$
6			q_0	q_1		$_1p_3$		SA	EA	EM
7					$_2p_1$		$_1p_4$	$+\bar{q}_1$	SA	EA
8	0			n_5		$_2p_2$		EM	$+\bar{q}_1$	SA
9		q_0		$_2p_1$	$_2p_2$		$_2p_3$	EA	EM	$+\bar{q}_1$
10			q_1	q_2		$_2p_3$		SA	EA	EM
11					$_3p_1$		$_2p_4$	$+\bar{q}_2$	SA	EA
12	q_0			0		$_3p_2$		EM	$+\bar{q}_2$	SA
13		q_1					$_3p_3$	-	EM	$+\bar{q}_2$
14	q_0	q_1	q_2	0	$_3p_1$	$_3p_2$	$_3p_3$	-	-	EM

Figure 12. Division example.

The conventions of the multiplication example, Section 3.3.2 will be used, together with the following additions

SA indicates that the corresponding DPU has transmitted its digit of the A register to the PCU to allow the PCU to select the next quotient digit to be used,

$_{j}p_{i}$ is the i^{th} digit of the j^{th} partial remainder,

q_i is the i^{th} quotient digit, and

n_i is the i^{th} digit of the dividend.

The arithmetic unit is prepared for the division at time 1. At time 2, the first quotient digit (q_o) is determined by the PCU and placed in the 'Pr Data' register. At time 3, the PCU issues an addition micro-instruction to DPU_1 which causes the first partial remainder to be determined. The 'Pr Data' register does not participate in this addition. A left shift M micro-instruction is issued next by the PCU. This causes q_o to be sent into the M register and causes n_4 to be placed in Pr Data . At time 5, a left shift A micro-instruction is launched by the PCU, causing n_4 to become the least significant digit of the A register. The cycle above is repeated* for each of the quotient digits.

---

*The final left shift of A is not performed.

4. INTERACTION WITH MEMORY

4.1 Introduction

The arithmetic structure must interact with a memory in order to obtain operands and to return results. The major parameter which determines the method of communicating between the DPUs and memory is the number of digits contained in the memory byte, where 'byte' is used in its generalized sense of operational data unit.

The methods applicable in two cases will be discussed; those for which the memory byte is one digit and those for which it is a number of digits. It is unnecessary to consider systems in which several memory bytes are required to represent one digit, because the elements of such a system would be structurally identical* to a system in which the memory byte is one digit.

We will assume that memory micro-instructions make up a small fraction of the micro-instructions executed by the DPUs and that the number of locations in the memory is very large.

---

*The major differences are the reduced width of the memory busses and the added complexity of the DPU caused by the requirement to make a number of memory accesses to execute one memory reference micro-instruction.

These assumptions lead to the conclusion that the memory address should not be carried as part of the micro-instruction stream. Instead, a pointer to a register may be sent along with the micro-instruction (as ${}_{j}F_{i}$ in Equation 2.1.1 through 2.1.3), since this would decrease the number of interconnections required to transmit this address information.

4.2 Methods Applicable When the Memory Byte Is the Digit

There are two methods of causing data transfers between the DPUs and memory when the memory byte is one digit of the number representation.

The simpler of the two methods, suggested by Comfort (8), is to cause all communication to take place via the PCU. The digits are placed in or removed from the DPUs by means of left shift micro-instructions. The left shift micro-instruction is defined so that the digit which will appear in the last DPU is fixed by the PCU when it issues the micro-instruction. To 'load' a register from memory, the PCU would alternately perform a read memory cycle and a left shift which causes the digit just read from memory to be the least significant digit of the register. To store a register, the PCU alternately performs left shifts and write memory cycles where the digit stored in the memory is the digit shifted out of DPU_1. A load and store of the same register can be

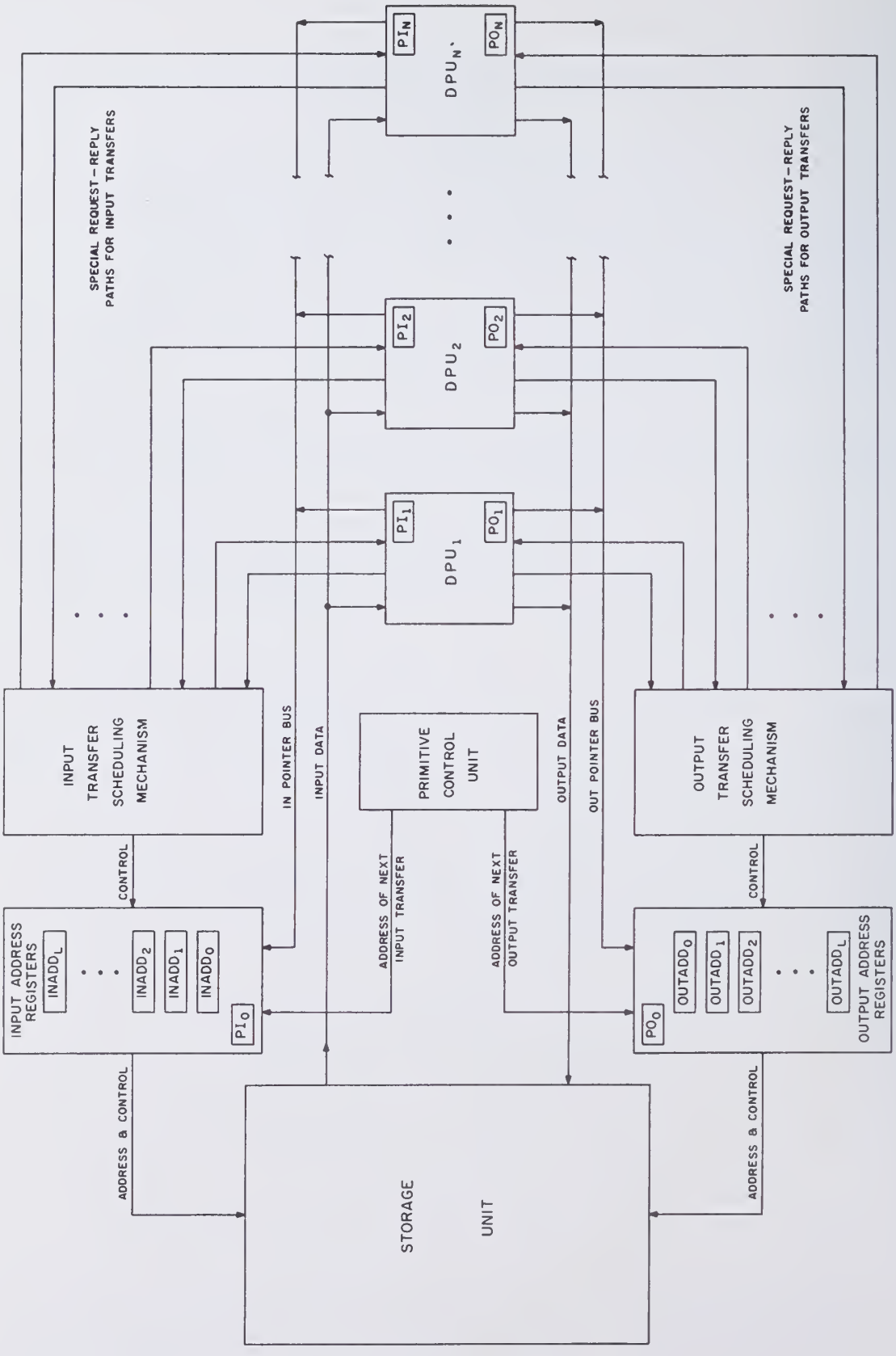


Figure 13. Communicating with a one digit per byte memory.

accomplished simultaneously. To do this, the most significant digit of the number to be loaded is obtained from memory. Then the left shift micro-instruction is performed such that the digit obtained from memory becomes the least significant digit. The digit shifted into the PCU by that micro-instruction is then stored in the memory as the most significant digit of the number being stored. The next digit of the number to be loaded is then obtained and the sequence above repeated. The sequence is repeated as many times as there are digits in the numbers.

The other method of loading and storing the arithmetic registers involves the use of generally distributed memory buses, sets of address registers, and scheduling mechanisms. There are two sets of each of these items, one for loading operands and one for storing results. This method requires only one micro-instruction to be issued per data transfer. The diagram of the system is presented as Figure 13.

The operation of the system is as follows. When the PCU detects a load or store operation, it transmits the memory address to the address register indicated by the free pointer register (PI_0 or PO_0). If there are fewer address registers indicated than DPUs the storage of this new address must be preceded by a check

that this address register is available*. If the register is free, the address is stored in the register. The pointer register is incremented by one, modulo the number of registers in the group. The appropriate micro-instruction is then issued to the first DPU by the primitive control unit. This DPU requests that its information digit be transferred to or from the storage unit. When the scheduling mechanism sends the signal to proceed, the DPU sends the value of the appropriate pointer register (PI_x or PO_x) to the address registers, which convert it to a memory address by a table-lookup process in the input address registers or output address registers (whichever is appropriate). It transmits or accepts the data digit, and passes the micro-instruction to its immediate neighbor, which then goes through the same cycle of events. The address which was referenced is also incremented to point to the location of the next digit of the operand.

Note that since all "store" micro-instructions prior to a given "load" micro-instruction are guaranteed to have been performed by a given DPU, there is no danger that an inappropriate value of a given variable is used by the adder. The choice which the designer has in interfacing a digit oriented storage device to

---

* The arithmetic unit must wait until it is available.

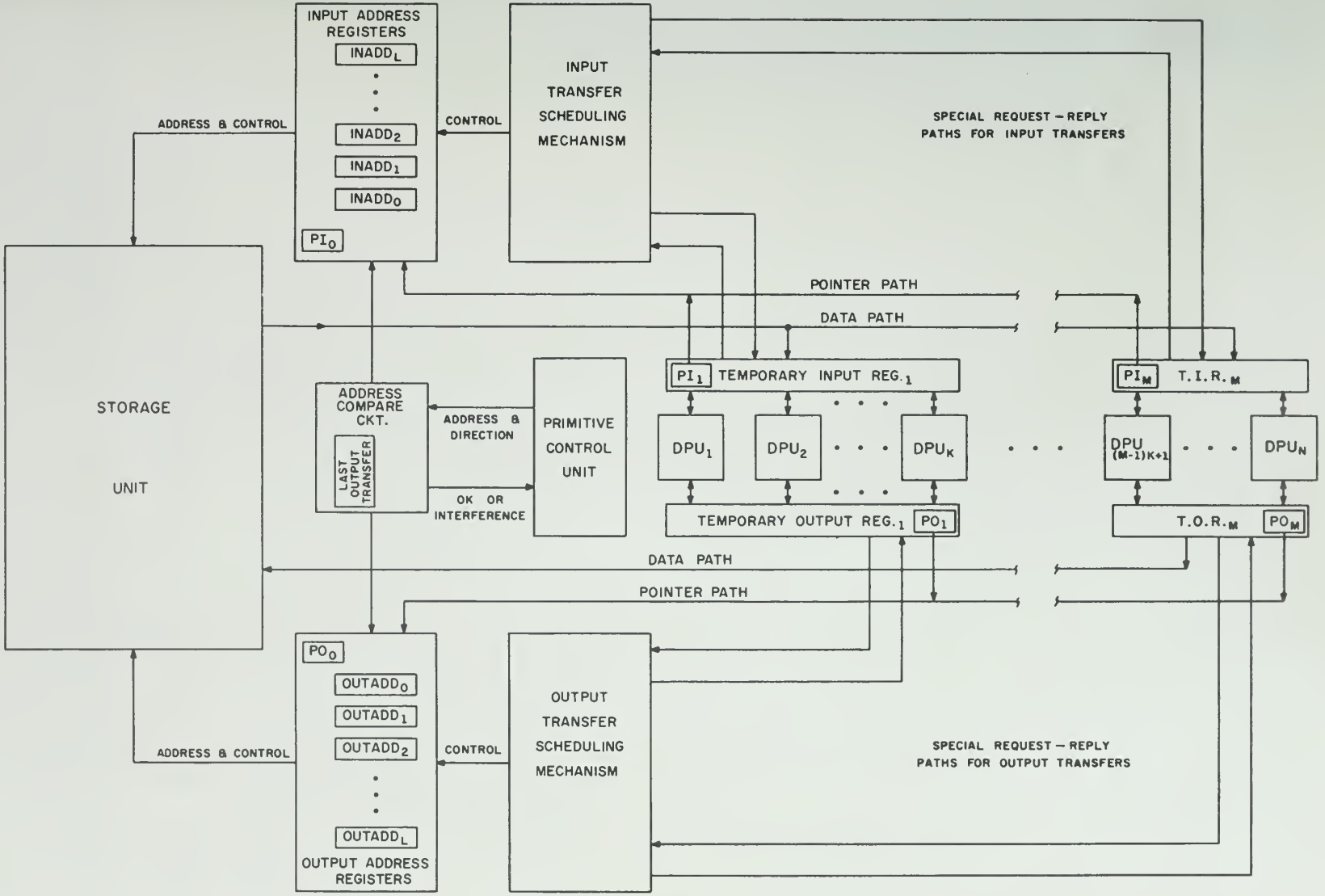


Figure 14. First method of communicating with a memory having a number of digits per byte.

the arithmetic unit is between a method which entails the minimum additional hardware and a method which may be faster but which requires an extensive amount of additional hardware. The second method is expected to be faster by approximately the number of digits contained in each operand.

4.3 Methods Applicable When the Memory Byte Is a Number of Digits

We will now discuss the methods of communicating between the arithmetic unit and memories which transfer a number of digits per transaction. One problem, not present when the memory byte is the digit, then faces the designer. He must incorporate serializing and deserializing mechanisms into the interfaces.

In the case of the first method discussed in the preceding section, in which all communication takes place through the PCU, the extension is very straight-forward. The serializer and deserializer simply become part of the PCU, since only one data transfer can be occurring at any given time in each direction.

The approach of employing a centralized serializer and deserializer does not appear to be appropriate when the method depicted in Figure 13 is extended. There are two methods of providing communication between the memory and the arithmetic unit in this case. The first method, shown in Figure 14, uses registers capable of storing one memory byte to convert data lengths.

Each of the registers is interfaced with as many DPUs as there are digits in the memory byte. When a load micro-instruction reaches the first DPU connected to a given input register, the control circuitry associated with the register cause the appropriate information to be put into the register*. Each DPU connected to the register then gates in its digit from the register when it executes the load micro-instruction. Each DPU executes a store micro-instruction** by storing the appropriate digit into its portion of the output register to which it is connected. When all of the positions of a register have been filled, the contents of that register is stored in the memory.

The storage address register scheme described in Section 4.2 can also be used with this method, except that now addresses are associated with buffer registers, not DPUs.

Since data which a given DPU presents to its output register for storage is not guaranteed to have been stored before it performs subsequent micro-instructions, a mechanism must

---

*The register is filled only if all of the DPUs connected to the register have executed the preceding load micro-instruction.

**The DPU must be inhibited from executing the store instruction until the preceding information has been sent to the store unit.

be included which assures that subsequent fetches will always retrieve the appropriate data. A simple method of guaranteeing this is to compare the address of the operand which is to be fetched with the address of the last store operation. If they are not the same, then obviously the memory will contain the most recently calculated data when it is requested. A given DPU cannot obey a store micro-instruction unless the store buffer register is assigned to collect data for that store operation, therefore implying that all prior store instructions have been completed.

If on the other hand, the address of the last store operation is the same as that of the operand, there is a possibility that for some DPU the store has not been made prior to the load. This condition can be controlled by means of an interlock. This interlock could consist of altering the micro-instruction from a storage unit reference to a reference to the contents of the output register*. This may necessitate the inclusion of register to register transfer micro-instructions not otherwise required. A

---

*This has some analogy to the Common Data Bus (24).

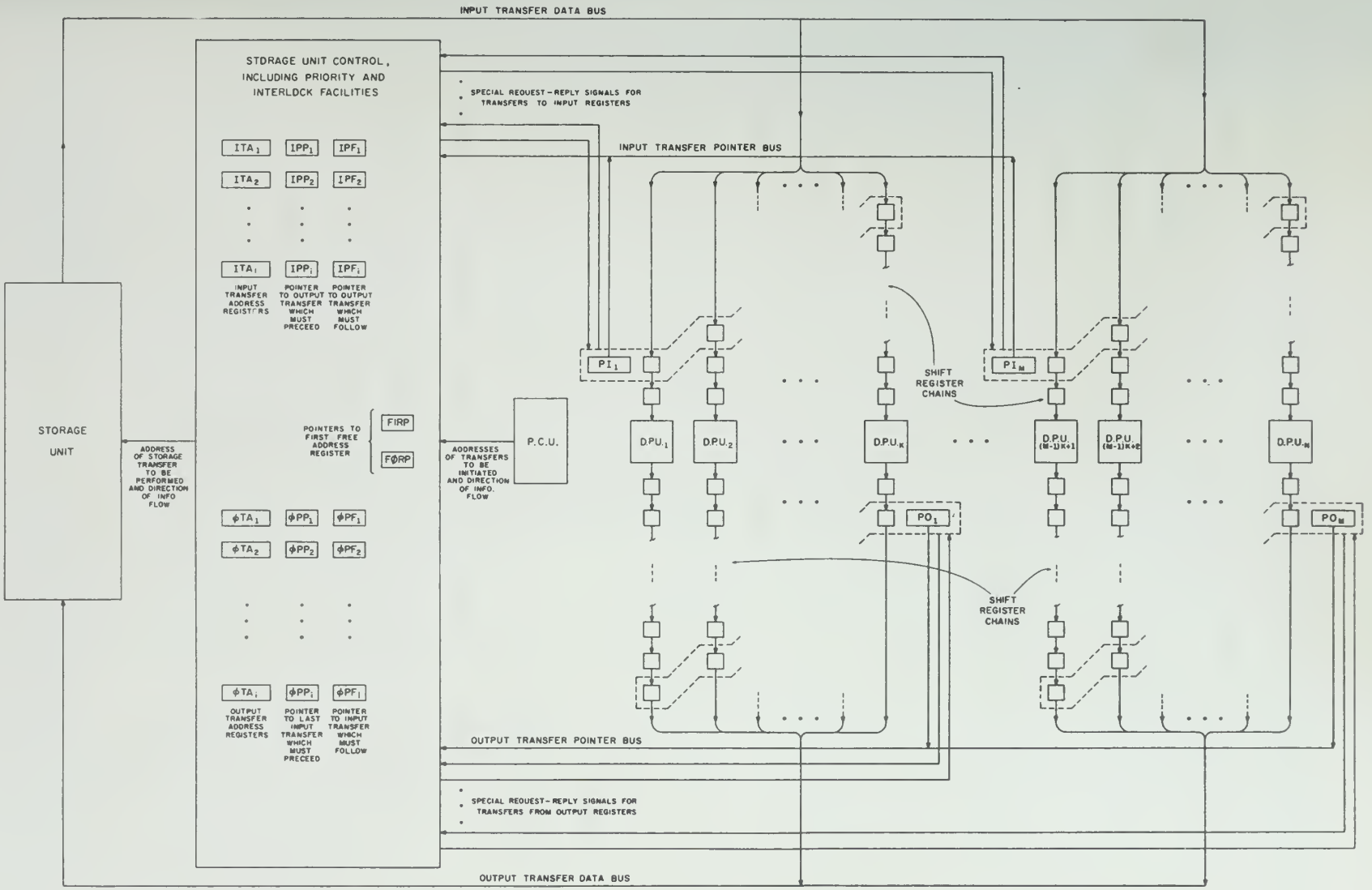


Figure 15. Second method of communicating with a memory having a number of digits per byte.

second method of implementing the required interlock consists of a micro-instruction which would not proceed past the first* DPU attached to an output register unless that register had stored all operands previously presented to it. This has the undesirable effect of requiring either two types of DPU or some way of altering the action of the universal DPU on the basis of whether it is a "first" unit or not. This method of byte length conversion has one distinct disadvantage. A second memory micro-instruction of the same type, i.e., the second of two loads or two stores, must not be allowed to be performed by the first DPU associated with a register until after the last unit associated with that register has completed the previous similar micro-instruction and the appropriate memory action has been taken.

If this causes intolerable delays, a second method of byte length conversion could be employed. This is depicted in Figure 15. In this approach the information exchanged between the adder and the memory passes through shift register-like structures. An element of this structure can store one digit and will accept data from its input neighbor when its indicator shows

---

*The most obvious micro-instruction meeting these conditions is a second store micro-instruction.

that it does not contain information. The element subsequently causes its own indicator to turn on and sends a signal to its input neighbor to turn that indicator off. These elements are arranged in chains; one extremity of each chain is connected to a DPU and the other is connected to a memory bus. There must be two chains associated with each DPU, one of which stages input operand digits, the other of which accumulates digits until a memory byte is collected. The former is used in loading operands from memory. The memory bus acts as the input neighbor of the first element and the DPU receives digits from the last element in each chain. The latter is used in storing results in the memory. The DPU is the input neighbor of the first element and the memory bus receives data from the last element of each chain. The shift register chains need not be equally long. For example, it is advantageous for the store chain associated with the last DPU of a given memory byte to be made smaller than the store chain receiving data from the first DPU of that storage byte. The chain of the first DPU must not only have room to store digits until the memory may be accessed but must store digits while the storage unit byte is being accumulated.

The address information can be handled very much like the methods above, viz. by a number of registers in which storage

addresses can be stored and pointers indicating the storage address register that each data register is to use.

More extensive checking is required to assure that the references to each specific memory location will be done in the order in which they were issued. Associated with each address register must be two dependency registers and a flag. The registers identify which other data transfers are also accessing this memory location, and the flag indicates whether or not the address register is currently in use. The registers that are used to indicate dependencies must have a null state, indicating no dependency. Both dependency registers point to address registers of the other type*. One points to the operation which must be completed (on a memory byte basis) before its operation may be performed. Registers of this type are labelled xPP_j in Figure 15, where j is the register number and x may be either I or Ø. I indicates that the register is associated with an input transfer, while Ø indicates output. The other dependency registers, labelled xPF_j, point to the last transfer found to be dependent on its transfer. These registers are used to eliminate unnecessary testing.

---

*That is, the dependency registers associated with a load point to address registers associated with store operations, while those associated with a store point to address registers associated with load operations.

When a load or store appears in the instruction stream, the memory address is sent to the storage unit control. When the address register which is indicated by the appropriate free register pointer is available, the memory address is placed into it. The PCU is then allowed to issue the appropriate micro-instructions to the DPUs. The memory address is also compared with those contained in the currently active address registers of the other type*. If there are any matches, the register number of the most recently initialized matching transfer is placed in the xPP_j of the transfer being initialized and the register number of the transfer being initialized is placed into the xPF_j register of this matching transfer. If there are no matches, null is placed into xPP_j. The xPF_j register is always set to null when a transfer is initialized.

When a data transfer is requested, the transfer indicated by xPP_j is checked to be sure that it has been performed at least to the byte of the current request. The transfer must not be allowed to take place until it satisfies the above condition.

With this scheme for interfacing the memory to the DPUs, it appears very desirable to pre-fetch operands.

---

*That is, the dependency registers associated with a load point to address registers associated with store operations, while those associated with a store point to address registers associated with load operations.

While it is possible not to initiate the request for the first memory byte until the PCU issues the appropriate command to the DPUs, this introduces an avoidable delay equal to the time required by the memory to service this request. Of course, when operand look ahead is employed and a branch dependent upon the arithmetic properties of a result is encountered, the device must either cease operand look ahead or fetch operands that may be required and find some way of discarding operands which are not required. A micro-instruction which causes the operand in the input shift-register adjacent to the DPU to be discarded accomplishes this function. This appears to be the only instance of non-productive activity occurring in a limited connection arithmetic unit. Unfortunately, non-productive activity takes place in the DPUs after the branch is resolved, in addition to the non-productive memory activity prior to its resolution.

5. OPERATIONAL SPECIFICATION OF THE MODULES

5.1 Introduction

The design parameters of a limited connection arithmetic unit determine the number of distinct types of modules required and the detailed specification of these modules. This chapter discusses each of the modules with particular emphasis of the effects of the following parameters:

1. the method of communicating with memory,
2. the number of quotient digits determined at each examination of the partial remainder,
3. the number of digits examined in normalizing number representations,
4. whether a push-down stack is included in the End Unit,
5. the method of performing addition and subtraction,
6. whether multiplier recoding is performed, and
7. the parameters of the number representation scheme.

Item 1, the memory communication scheme, determines whether special modules, special data and control paths, and special micro-instructions will be included for implementing the communications paths between the arithmetic unit and memory.

This area was discussed in detail in Chapter 4. The number of registers in the arithmetic unit for holding intermediate results is also dependent on the scheme for interfacing the arithmetic unit and memory. Items 2 and 3, the number of quotient digits per examination and the normalization parameter, will determine the number of additional* DPUs which must have a direct data path to the PCU, and if special modules are required to signal the PCU when information is available on these paths. Item 4 determines the complexity of the End Unit. Item 5 determines the number of DPUs with which each DPU communicates. Item 6 determines whether the complexity of the PCU is increased by containing a multiplier recoder within it. Item 7, the parameters of the number representation, determines the size of all data registers and data paths. The method of performing arithmetic, item 5, also affects the size of the inter-DPU data paths.

---

*The DPUs other than those which will have a direct data path to the PCU because of its role as DPU_0.

5.2 The Digit Processing Unit

5.2.1 The Role of the DPU

The DPUs collectively perform the fractional part processing of the arithmetic unit. Each DPU contains one digit of each of the active operands; the significance of the digits contained in a DPU is inversely related to its 'distance' from the PCU, as shown in Figures 1 and 2. Each DPU contributes to the processing by executing a sequence of micro-instructions. The sequences executed by the DPUs of an arithmetic unit are identical except for minor substitutions or omissions*. A given micro-instruction in the sequence is executed by the DPUs from left to right (i.e., DPU_1, DPU_2, ..., DPU_n). At any given time there are a number of micro-instructions being processed by the DPUs.

---

*Recoding and 'transmit digit to PCU' micro-instructions will, in general, be replaced by a no-op or removed from the sequence after being executed by some of the DPUs.

5.2.2 DPU Registers

Each DPU retains one digit of each of the active operands. There must be at least three registers distributed through the DPUs: an accumulator register, a multiplier-quotient register, and an operand register. Each DPU must also contain a register to hold the micro-instruction it is executing. It must also have a serial number counter if the arithmetic unit employs request-response signals. This serial number is transmitted with the inter-DPU data and identifies the micro-instruction to which the data is to be applied. Additional registers may be distributed through the DPUs. One possible use of such registers is to hold the number temporarily displaced from the accumulator while the accumulator registers are used to shift the number that had been residing in an operand register that cannot be shifted. Only one register need be included for this purpose. A second use of such registers is to hold intermediate results which are needed so soon after they are calculated that storing them and retrieving them from memory would delay the processing. The number of intermediate result registers that are desirable to include for this purpose is dependent upon the method of communicating between the arithmetic unit and memory. The number is determined by trade-off considerations. The decrease in the average time spent waiting for operands to

become available must be compared with the additional hardware required to cause the decrease.

5.2.3 The Micro-Instruction Repertoire

There are a number of micro-instructions which may be included in the repertoire of the DPUs in addition to those discussed in Section 2.3 and Chapter 4.

The first of these micro-instructions causes the digits of a specific register to be placed on the inter-DPU data paths. These micro-instructions would be used during normalization and division. With one of these micro-instructions the PCU can obtain the information it required to determine what additional processing is required to complete the operation. While the left shift micro-instruction may be used for this purpose, special micro-instructions have advantage in some designs. For example, modules which determine when the information required by the PCU is available are much simpler when special micro-instructions are used.

The second of these micro-instructions causes one of the normalization recodings (Equation 3.5.3 and 3.5.4) to be performed. These micro-instructions make it possible to reduce the number of shifts required to perform a recoding. When the recoding micro-instructions are not implemented and a number

must be recoded, all the digits to be changed must be shifted into the PCU. All but the leading zero digits of the recoded number must then be shifted back into the DPUs.

When the normalization recoding micro-instructions are implemented, a normalization recoding is begun by shifting out all digits which are to be recoded into leading zeros. The recoding micro-instruction is then issued to DPU_1, which passes to DPU_2 the indication that a recoding micro-instruction is to be performed. DPU_2 then indicates by the value of G_2 whether it will participate in the recoding*. DPU_1 recodes its digit when it receives this information. DPU_2 then passes to DPU_3 the indication that a recoding is to be performed. DPU_3 responds by indicating whether or not it will participate in the recoding by the value of G_3 which it sends to DPU_2. DPU_2 then recodes its digit. Each successive DPU goes through this process until the micro-instruction reaches the first DPU which cannot recode its digit. After this DPU sends its response to its left neighbor, it terminates the recoding by passing either no micro-instruction or a no-op micro-instruction to its right neighbor.

---

*This is necessary because the recoding of the last digit is different from the recoding of all the other digits.

A no-op and an execute micro-instruction must be included in the repertoire of the DPUs if they are designed to execute micro-instructions periodically and synchronously. No-op micro-instructions must be executed prior to any micro-instruction which requires information form neighboring DPUs. The number of no-ops which must be given after the set-up micro-instruction is equal to the number of DPUs that must send information to the DPU executing the micro-instruction. These no-ops are followed by an execute micro-instruction. Note that if each DPU saves the information about the last micro-instruction it has set up, only one execute micro-instruction is necessary.

The last of the additional micro-instructions that a designer may wish to include in the repertoire of the DPU is one that places a constant, such as zero, in a register. The representation of the constant determines the details of the micro-instruction. In the most useful cases, where all of the digits of the constant are identical*, the DPUs do not have to cooperate with any of their neighbors in performing the micro-instruction. The value of the digit may be implicit in the micro-instruction or it may be sent as the modifier value sent along with the micro-instruction.

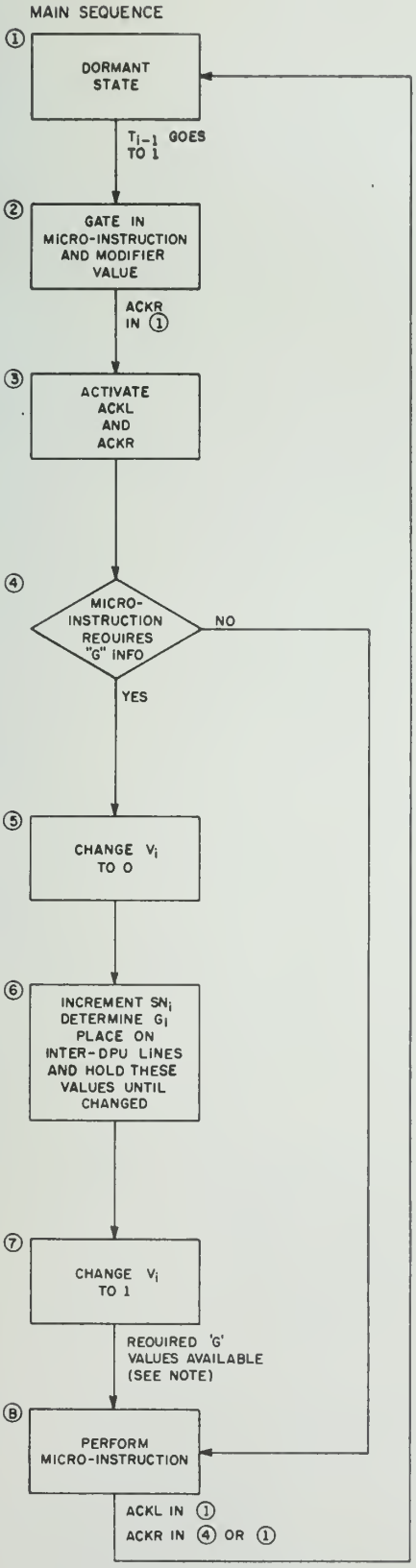
*The most important examples of numbers whose digits are identical are zero, the largest number, and the smallest number.

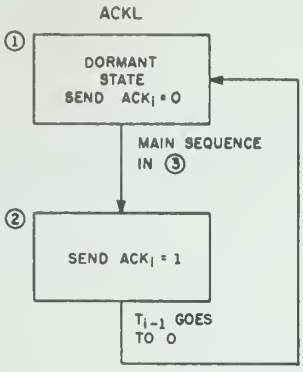
5.2.4 The Sequencing and Coordination of the DPUs

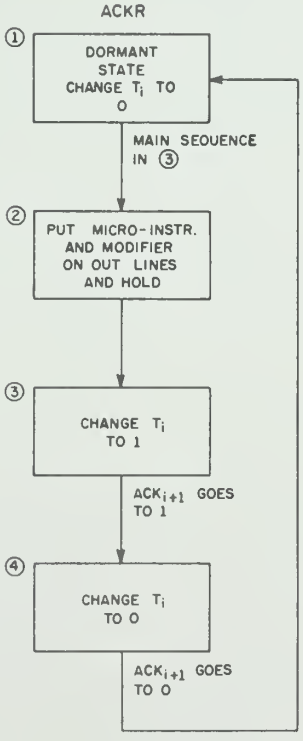
The operation of the DPUs contained in an arithmetic unit must be coordinated to obtain useful results. Each DPU must execute the same sequence of micro-instructions, which is determined by the processing to be performed and the specific operand values. After executing micro-instruction j-1, a typical DPU, DPU_i, must determine the value of ${}_jG_i$ and place this value on its inter-DPU data lines (see Equations 2.1.1 through 2.1.3). This information is required by $DPU_{i-\alpha_j}$, ..., DPU_{i-2}, and DPU_{i-1} to perform micro-instruction j. DPU_i also passes the j^{th} micro-instruction and modifier to DPU_{i+1} so that DPU_{i+1} will determine ${}_jG_{i+1}$. When DPU_i receives ${}_jG_{i+1}$, ..., ${}_jG_{i+\alpha_j}$ it performs micro-instruction j and begins the procedure for micro-instruction j+1.

The method of achieving this coordination is determined by the method of implementing the DPUs. We will discuss how this coordination may be achieved for two implementations which are at the extremes of implementation philosophies.

In the first method of implementing the arithmetic unit, all of the DPUs periodically and synchronously execute micro-instructions. Each DPU goes through the same two cycle operation. On the first cycle each DPU executes the micro-instruction it has just received. On the second cycle each DPU







NOTE REQUIRED 'G' VALUES ARE AVAILABLE TO DPU_i WHEN ALL DPU'S WHICH MUST SEND INFORMATION TO DPU_i INDICATE THAT THE INFORMATION THEY SEND ARE VALID ($V_x=1$) AND SEND A SERIAL NUMBER EQUAL TO THE SERIAL NUMBER COUNTER OF DPU_i ($SN_x = SN_i$). THAT IS, THE FOLLOWING CONDITION MUST BE SATISFIED:

$$\prod_{e=i}^{a_i} \left[(V_{i+e}=1) \cdot (SN_{i+e}=SN_i) \right] = 1 \text{ (TRUE)}$$

Figure 16. Flow diagram of the control logic of DPU_i.

simultaneously passes the micro-instruction it has just executed to its right neighbor and receives from its left neighbor the micro-instruction which that DPU has just executed.

Coordination of the DPUs can be accomplished by the use of no-op micro-instructions. Each micro-instruction which requires the cooperation of several DPUs is preceeded by a set-up micro-instruction and a number of no-op micro-instructions. These micro-instructions assure that the required information is available when a DPU executes the micro-instruction.

At the other extreme of implementation philosophies are the arithmetic units in which all activities are controlled by request-response signals. The control logic of each DPU may be composed of three interacting sequential machines in this case. The flow diagrams of these machines are shown in Figure 16, where the control signals are:

T_i which indicates that the lines between DPU_i and DPU_{i+1} which carry the micro-instruction operation code and modifying data $_jF_i$ are valid,

ACK_i indicates that DPU_i has accepted the micro-instruction from DPU_{i-1},

SN_i is a serial number which indicates the micro-instruction for which the $_jG_i$ data was determined,

and

V_i indicates that the $_jG_i$ and SN_i lines are stable and may be examined.

All of the sequential machines of all DPUs are initialized to their respective state 1 in the figure. The V_i signals are initialized to 1, the T_i and ACK_i signals are initialized to 0.

DPU_i begins the execution of a micro-instruction when it is in its dormant state and T_{i-1} goes to 1. This indicates that DPU_{i-1} is transmitting to it the next micro-instruction it is to execute. DPU_i gates this micro-instruction into its micro-instruction register and activates the ACKL and ACKR machines. ACKL indicates to DPU_{i-1} that it has received the micro-instruction, while ACKR passes the micro-instruction to DPU_{i+1}. If DPU_i must send information to other DPUs which they require to execute this micro-instruction, it turns off V_i. It then increments the serial number, SN_i, determines the required information and places the information and identifying serial number on the inter-DPU data paths. DPU_i then turns V_i back on and begins checking the inter-DPU data paths for the information that it requires to perform the micro-instruction. When this information becomes available or if information from other DPUs is not required, DPU_i performs the micro-instruction (i.e., it changes the value of some operand digit it contains).

When the micro-instruction has been performed, ACKL is in the dormant state, and ACKR is in state 4 or the dormant state, the main sequence goes into the dormant state to wait for the next micro-instruction.

5.2.5 The Number of Connections to a DPU

Figure 3 and the discussion above may be used to determine the number of electrical connections (pins) that must be made to each DPU. In an arithmetic unit which uses request-response signals this number is

$$C_{RR} = P + 4 + 2\,(OPS+SF) + (\alpha+1)\,(SG+SSN+1) + MEM \qquad (5.2.1)$$

where

C_{RR} is the number of connections required by each DPU when request-response signals are used,

P is the number of pins required to power the DPU,

OPS is the number of bits required for the operation code of the micro-instruction,

SF is the number of bits of the modifier value accompanying the micro-instruction,

SG is the number of bits required to represent a ${}_{j}G_{i}$ value,

SSN is the number of bits in the serial number, and

MEM is the number of pins required by the DPU to communicate with memory.

If we assume that the only values of ${}_jF_i$ and ${}_jG_i$ are those required to perform addition and shifts, SF and SG are then

$$SF = 1 + Lg\,(r-\ell+1), \text{ and}$$

$$SG = Lg\,(2r(r-\ell)+1),$$

where

r is the radix of the number system,

ℓ is its redundancy parameter, and

Lg(x) is the smallest integer equal to or greater than $\log_2(x)$.

If we further assume that the PCU does not have any unusual data requirements so that SN has to take on α values or SSN = Lg (α), (5.2.1) becomes

$$C_{RR} = P + 6 + 2 \cdot OPS + 2 \cdot Lg\,(r-\ell+1) + (\alpha+1) \cdot \left[Lg\,(2r(r-\ell)+1) + Lg\,(\alpha)\right] + MEM \qquad (5.2.2)$$

When the arithmetic unit is implemented so that all DPUs execute micro-instructions periodically and synchronously, the request-response signals and serial numbers are unnecessary. Synchronizing signals are required instead; the number of connections then becomes

$$C_s + P + 2 \cdot (OPS+SF) + (\alpha+1) \cdot SG + SYNC + MEM \qquad (5.2.3)$$

where

C_s is the number of connections required by each DPU in a synchronous arithmetic unit, and

SYNC is the number of connections required to transmit synchronizing signals to a DPU.

If the same assumptions are made regarding the values taken on by ${}_jF_i$ and ${}_jG_i$ as was made above, this becomes

$$C_s = P + 2 + 2 \cdot OPS + 2 \cdot Lg\,(r-\ell+1) + (\alpha+1)\cdot Lg\,(2r(r-\ell)+1) + SYNC + MEM. \qquad (5.2.4)$$

5.3 The Primitive Control Unit

5.3.1 The Role of the PCU

The main function of the PCU is to convert a sequence of instructions (e.g., add, multiply, divide) into a sequence of micro-instructions which must be performed by the DPUs and to issue these micro-instructions to DPU_1. This conversion process is similar to the process that the adder control logic of the IBM 7094(11) performs in interpreting instructions. The major difference between the two processes is that the multiplication algorithm must be right-directed in the limited connection arithmetic unit, as discussed in Section 3.3.

The PCU may also perform subsidiary functions, such as processing exponents or communicating with the memory.

5.3.2 The Registers in the PCU

The PCU must contain an instruction register which contains the instruction currently being converted into micro-instructions. It also contains the integer extension of the accumulator, which must be three digits in length (see Section 3.3). The PCU must also have a counter to control the number of shifts and the number of repetitive steps during multiplication and division.

If the arithmetic unit is implemented with request-response signals and 'sense micro-instruction' detectors are not used, the PCU must contain a serial number counter identical to those in the DPUs. The PCU must be able to store the value of several multiplier digits if it recodes multipliers. Memory byte assembly and dis-assembly registers are required in the PCU of arithmetic units that communicate with memory through the PCU and in which the memory byte contains several digits.

The exponents of all operands whose fractional parts are contained in the DPUs may be contained in and process by the

the PCU to obtain higher performance. It is possible to process exponents in a special module or in a second set of DPUs, but these choices tend to increase the time between when the manipulation of the exponents is begun and when its result is available. Since the result of the exponent manipulation determines the processing to be performed on the fractional parts of the operands, such delays decrease performance.

5.3.3 The Sequencing of the PCU

The PCU is sequenced very much like the DPUs are (see Figure 16). The PCU does not have an ACKL machine, since it has no module to its left. When the PCU executes a micro-instruction it is primarily determining the next micro-instruction that it must issue to DPU_1.

5.3.4 The Connections to the PCU

The PCU must communicate with three major components of the computer system. The first of these is the complex of DPUs; these connections are shown in Figure 1. In arithmetic units where the PCU requires from the DPUs only that information which it gets because it is in effect 'DPU_0', the number of pins required by the PCU for this information is less than the number required by a DPU. The PCU requires one, rather than two, sets of connections

for issuing micro-instructions. It also requires one fewer set of inter-DPU signals ($_{j}G_{i}$, SN_{i}, and V_{i}) in this case since it has no left neighbor.

If exponents are processed external to the PCU, some connections will be required to indicate what processing is required and to return an indication of the results of the processing. If a special module is employed to process exponents, the number of connections required by the PCU to communicate with the special module can be kept small. If a second complex of DPUs is used to process exponents, the number of connections can be expected to be quite large.

The second component of the computer system with which the PCU must communicate is the memory unit, from which it obtains instructions and possibly obtains operands and returns results. The number of pins required is determined by the memory unit byte, the number of bits required to address the memory, and whether results must be stored.

Finally, the PCU must be connected to power sources and possibly synchronizing signals. The number of pins required for this purpose is the same as the number of pins required by a DPU.

5.4 The End Unit

The End Unit performs two major functions in the arithmetic unit. The first function is to act like a terminator for the DPUs. Terminating a set of DPUs with an End Unit is analogous to terminating a transmission line with its characteristic impedance: the DPUs operate as if DPUs extended indefinitely to the right. The End Unit supplies all of the signals and data that are required by the DPUs but are not supplied by other DPUs or the PCU.

The second function of the End Unit is to cause all calculations to be performed with the maximum possible precision. It does this by saving the digits shifted out of the last DPU in one or more push-down stacks. These digits can then be returned to the DPUs when left shifts are given or be used in forming sums.

The complexity of the End Unit is essentially independent of the complexity of the DPUs or the PCU. The End Units of arithmetic units designed with request-response signals must have a serial number counter. If an End Unit is used to increase the precision of calculation it must have a micro-instruction register in addition to the push-down stacks for holding the digits shifted out of the DPUs. The number and capacity of the push-

down stacks are determined by economic considerations. In the simplest scheme, the End Unit has one stack which is associated with the A register. The Ø register cannot have shifting capability in this case.

The End Unit requires fewer pins than either the PCU or the DPUs. Not only does the End Unit need only one micro-instruction bus, but it needs only one validity signal and one serial number bus. The End Unit also requires one less inter-DPU data bus. The connection to the End Unit of a typical arithmetic unit is shown in Figure 1.

The control of the End Unit in an arithmetic unit employing request-response signals is very much like the control of a DPU, shown* in Figure 16. It does not require the ACKR machine since it does not have any right neighbors.

---

*In Figure 16, i = n+1 for the End Unit. The End Unit must supply multiple V_i, SN_i, and ${}_jG_i$ signals. We will define that $V_{n+1} = V_{n+2} = \ldots = V_{n+\alpha}$ and $SN_{n+1} = SN_{n+2} = \ldots = SN_{n+\alpha}$ and will interpret ${}_jG_{n+1}$ to read ${}_jG_{n+1}$, ${}_jG_{n+2}$, ..., ${}_jG_{n+\alpha}$. α is the maximum of the α_j of the micro-instruction repertoire of the DPUs.

5.5 Exponent Arithmetic Unit

There are three methods of processing exponents in a limited connection unit. The first method is to process the exponents in a second complex of DPUs. This method will result in a low performance arithmetic unit and will require the PCU to have a large number of electrical connections. The second method is to perform exponent processing in the PCU.

This method makes it possible to obtain higher performance and to decrease the number of pins required by the PCU. It, however, increases the complexity of the PCU by requiring the exponents of all active operands to be stored in the PCU.

The third method, employing a special module to hold and process exponents appears to make it possible to achieve high performance while not increasing the complexity or the number of pins of the PCU excessively.

The Exponent Arithmetic Unit would best be designed to send to the PCU only the information that it requires to control the processing of the DPUs. For example, if an addition is to be performed, the Exponent Arithmetic Unit would respond with one of the following:

a) shift A register,

b) shift Ø register,

c) issue add micro-instruction.

The Exponent Arithmetic Unit must have commands to set up multiplication and division in addition to those which have analogies to the micro-instructions of the DPUs. These set up commands cause the exponent of the result to be determined and the repetitive step counter to be set. The PCU would then precede* each of the repetitive steps with an inquiry of the Exponent Arithmetic Unit, which would indicate whether another repetitive step is to be performed or if the operation has been completed.

5.6 The Sense Micro-Instruction Detector

Another PCU function which may be performed external to the PCU is that of determining when the necessary information is available from the DPUs. This function may be performed within the PCU just as it is within each of the DPUs. However, performing some or all of this function in other modules reduces the complexity and the pin requirements of the PCU. The unit that detects micro-instructions which were

---

*Each inquiry could be overlapped with the previous step of the algorithm.

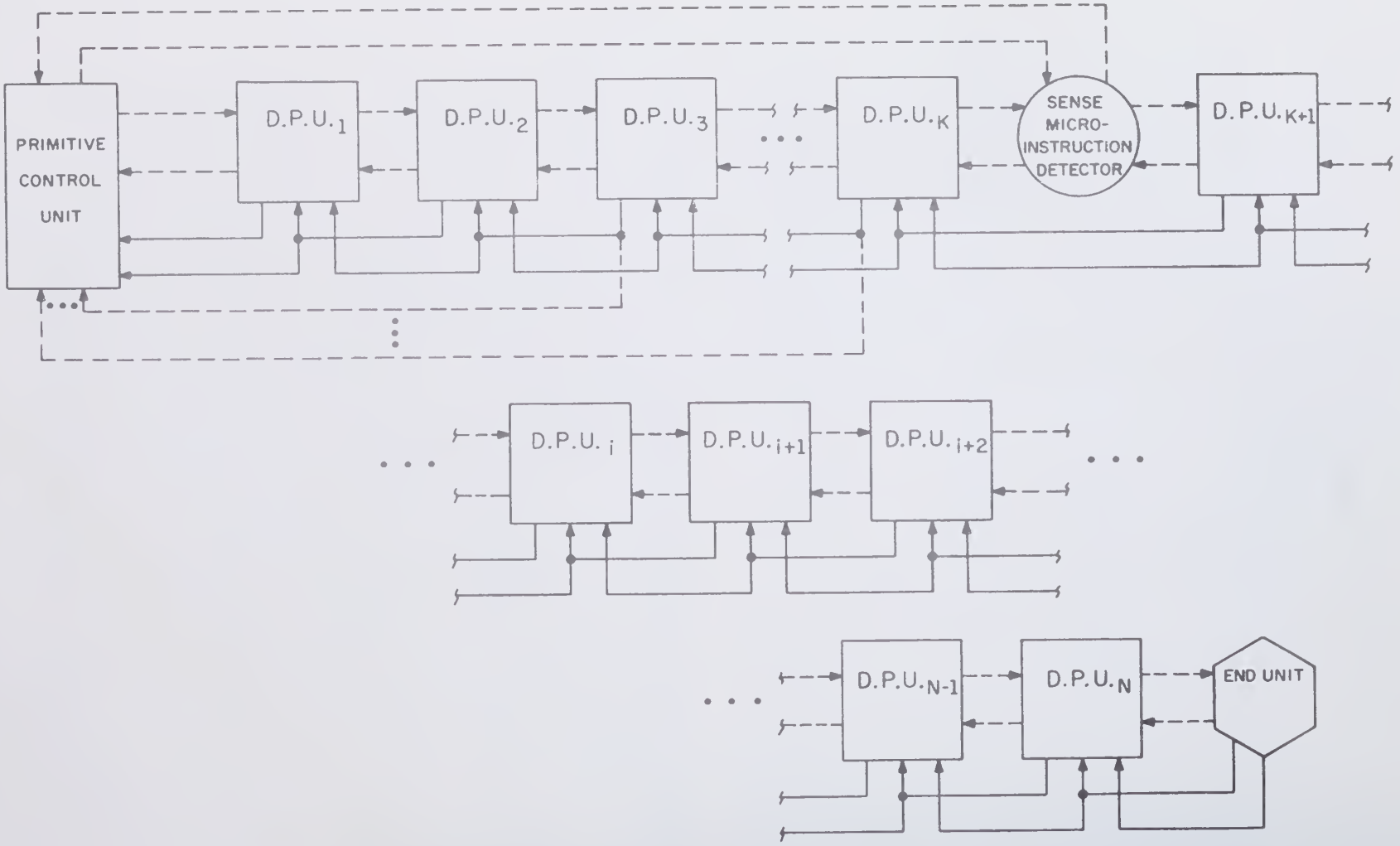


Figure 17. Placement of the sense micro-instruction detector in a typical arithmetic unit.

(or could have been) issued to examine the digits of one of the operands is placed between the last DPU which supplies information to the PCU and the first DPU which does not, as shown in Figure 17. When one of these micro-instructions reaches the sense micro-instruction detector, this unit sends a signal to the PCU indicating that it has reached the detector. Hence, it indicates to the PCU when the required information is available. The PCU then does not have to receive the serial numbers associated with the data it requires.

If unique micro-instructions are employed for this sensing operation, this simple scheme will suffice. Any micro-instruction which causes operands to be placed on the inter-DPU data connections may be employed to sense operands contained in the DPUs, however. Some mechanism must be included in this case to distinguish between the instances when these micro-instructions are being used to sense an operand from when they are not. The basic method employs a counter which contains the count of the subsequent 'sense-type' instructions which were not launched for sense purposes. There are two possible implementations of the sense micro-instruction detector when a counter must be included.

In the first, shown in Figure 18, the counter is located in the PCU; the detector module is little different from that used if unique micro-instructions were employed for sensing DPU contents. The major difference is that a second sense-type micro-instruction will not be allowed to propagate beyond the sense micro-instruction detector until a proceed signal is received from the PCU. This proceed signal indicates that the last sense micro-instruction has been tallied on the counter.

In the second, shown in Figure 19, the counter is in the detector. This has the advantage of minimizing the amount of hardware in the PCU. It also is less likely to cause delays because the signal to increment the counter is sent at the same time the micro-instruction is launched. Hence, incrementing the counter is overlapped with the micro-instruction propagating through the DPUs.

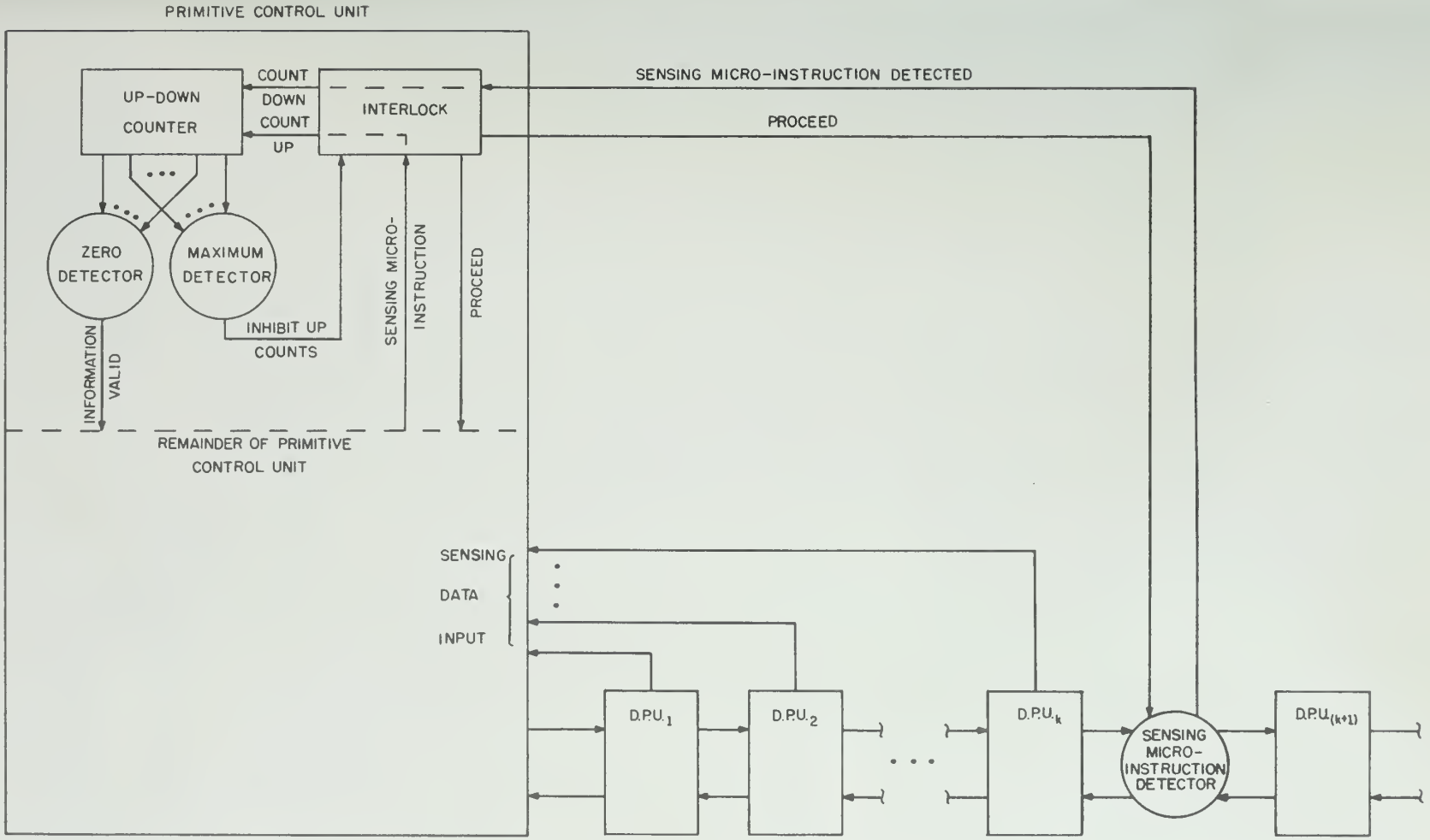


Figure 18. Counter associated with sense micro-instruction detector is located in the PCU.

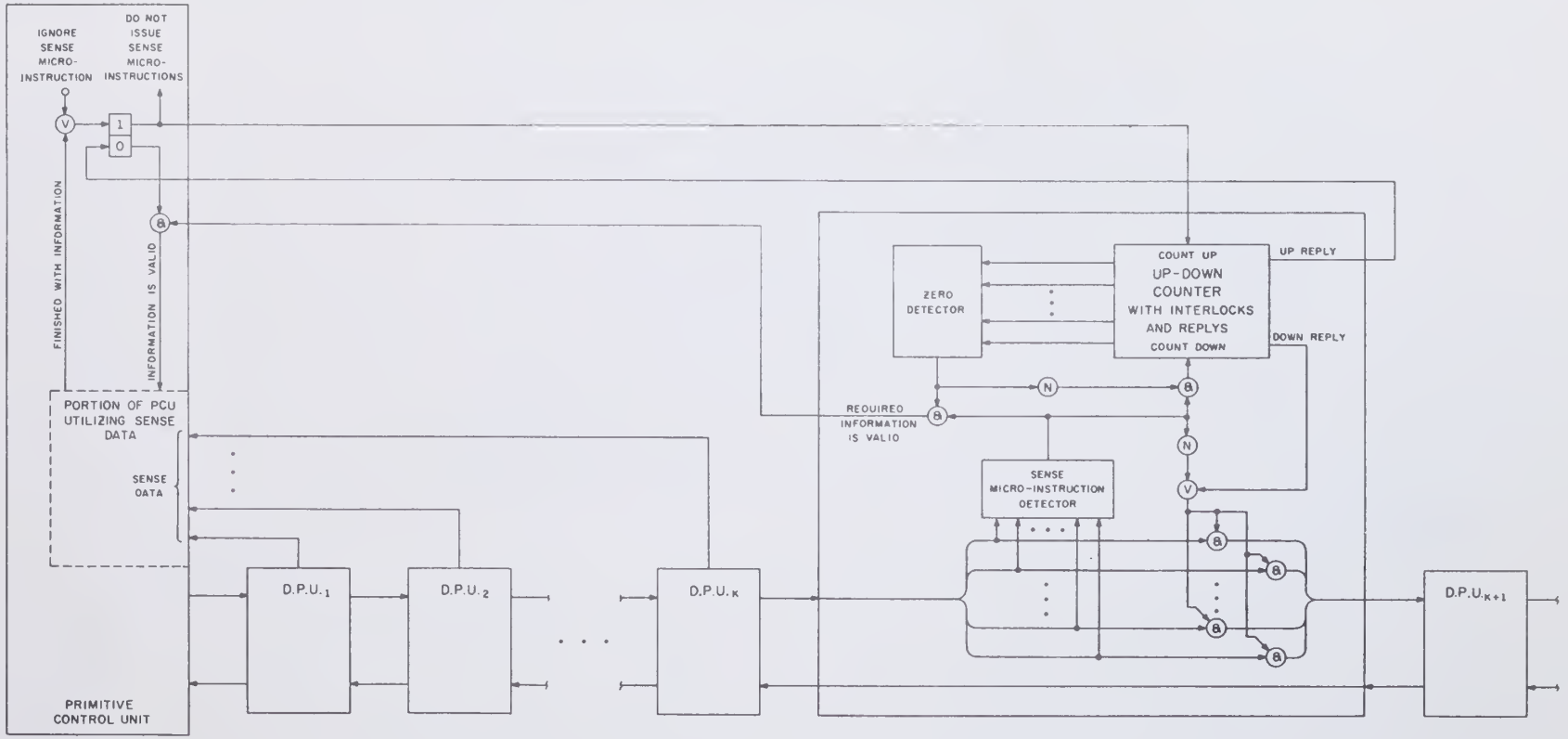


Figure 19. Counter is located in the sense micro-instruction detector.

6. SUMMARY AND CONCLUSIONS

6.1 Discussion of Results

Section 1.1 describes the characteristics that a mechanism must have in order for it to be particularly appropriate for implementation in the newly emerging technologies commonly called 'large scale integration' or 'LSI'. This paper presents a design for an arithmetic unit which admirably meets those requirements.

One of the primary desiderata of a unit to be implemented in LSI is that the item be composed of a small number of rather complex module types (i.e., the modules employ a large number of logic elements). The approach proposed in this paper will yield designs that will consist of from three to eight types of modules. Furthermore, the complexity of these modules can be adjusted over a wide range by such factors as the number system and algorithms used, so that a design may be tailored to the technology with which it is to be implemented.

Another major desideratum of designs to be implemented in LSI is that each module must require a relatively small number of connections (i.e., pins) to communicate with its environment.

The number of connections required by the modules of a limited connection arithmetic unit are limited by the need of each module to communicate with a small number of other modules. The major module, the Digit Processing Unit, must communicate with either two or four* other modules**. The End Unit must correspondingly communicate with one or two other modules. The number of modules that communicate with the Primitive Control Unit is no less** than the number that communicate with the End Unit.

A third desideratum of designs to be implemented in LSI is that no inter-module signal be routed to a large number of modules. No inter-module signal in arithmetic units organized as proposed in this paper need be sent to more than three modules; the number may be decreased to two or one with attendant decreases in performance.

---

*This case requires less than 50% more pins because of the nature of the signals.

**Arithmetic units in which the modules communicate with a larger number of other modules can be expected to have higher performance.

Hence, the approach proposed in this paper is very well suited to designing arithmetic units in LSI. Furthermore, the proposed organization has several advantages over the other arithmetic unit organizations proposed for implementation in LSI (9, 10, 12) because it is operationally a single multi-purpose arithmetic unit rather than a multiplicity of special-purpose units.

The first advantage is that no effort is required of the programmer, compiler, assembler, or monitoring program to take full advantage of the potential parallelism of the arithmetic unit. The arithmetic unit is organized so that the operations within a single instruction stream that can be performed concurrently are evoked automatically.

The second advantage of the proposed scheme is that no unnecessary processing is performed when a conditional branch is encountered in the instruction stream. After the controlling element (the PCU) is presented with a conditional branch instruction, it can immediately initiate the testing required to resolve the branch. Computers with multiple execution units may have to delay the testing until the operand is available. While this operand is being formed by one of the execution units and is unavailable for testing, the ins truction decoder either waits (with the attendant loss of performance), or goes into a

mode in which it continues to decode instructions and issues them conditionally to the execution units. The execution units are allowed to proceed but are not allowed to make any irrevocable changes to the state of the computer until the branch decision is made. This requires a great deal of interlocking and control hardware.

Like the other organizations proposed for implementing arithmetic units in LSI (9, 10, 12), the arithmetic unit proposed in this paper has the property of modularity. That is, the word length is determined by the number of modules and not by the design of the modules. The same basic building blocks could therefore be used to construct a variety of arithmetic units with a wide range of computational capability.

6.2 Suggestions for Related Work

The number system and arithmetic algorithms are the principal factors with which the design of a limited connection arithmetic unit may be adjusted to a specific technology and to a specific performance requirement. The designer has no general analytic tools to aid him in making these choices. He must therefore go through an iterative process of selecting tentative parameters, designing an arithmetic unit based on these para-

meters, and evaluating the desirability of the resulting design. Studies which would be particularly useful to the designer of limited connection arithmetic units include:

1. the determination of the complexity of a signed-digit adder as a function of its number system,
2. the determination of the improvement in performance that will occur if multiplier recoding is employed or if additional digits of the partial remainder are examined during division,
3. the determination of the complexity of multiplier recoders, quotient digit selectors, and normalization control circuitry.

Reliability and availability considerations were not addressed in this paper. The arithmetic unit as proposed in this paper will operate properly only if all the modules are operating properly. The modules can be designed so that they will stop if they detect an error, so that maintenance becomes less burdensome. Determining organizational modifications that are necessary to yield an arithmetic unit that will operate properly in the presence of failures is a very important area to be investigated.

Another unresolved problem in the area of the organization of the limited connection arithmetic unit is the provision of a reasonable method of performing multiple precision addition and subtraction. The method which would have to be used in the arithmetic unit as described here requires very many shift micro-instructions to be performed whenever radix point alignment is necessary. The digits shifted out of the active adder register during radix point alignment are not shifted into another active register, as they are in the classical Von-Neumann arithmetic unit. Instead, they are shifted into the End Unit, from which they can be returned to the active registers only by left shifts. Reconstructing these digits from the initial operands would also require a significant amount of shifting.

A suggestion for improving the performance of the limited connection arithmetic unit was recently made by Robertson (20). He observed that more optimal normalization and quotient digit selection algorithms could be employed if each zero digit is assigned the sign of the first non-zero digit to its right. For example, if a number to be normalized has the form 10 ... 01, it would be clear that the number is normalized after examining the first zero digit, whereas all of the zero digits would have to be examined and shifted if the zero digits were unsigned. Organizing the mechanism for associating

signs with zero digits so that the appropriate zero digit receives the sign is the fundamental problem to be solved in applying this technique to limited connection arithmetic units. If a sum has a number of adjacent zero digits, the DPU containing the first of these zero digits will have performed a number of micro-instructions before the sign to be associated with that zero digit arrives at the DPU. These micro-instructions may have made additional copies of the zero, sent it to another DPU, or obliterated it. Developing a scheme for keeping the appropriate records with a reasonable amount of hardware appears to be a challenging problem.

APPENDIX I

CHARACTERISTICS OF THE SYMMETRIC RADIX TWO SIGNED DIGIT ADDER

The probability of each possible pair of adjacent digits in the sum was determined for Adder 3 of Section 3.2. The analysis is based on the assumption that one input representation, A, has the same pair probabilities as the sum representation, A', while the other input, $\emptyset$, is characterized parametrically by an analogy to SRT division (19), (21). The analysis was performed for the case for which the probability of zero for a digit in the parametrically defined input representation was in the range $\frac{2}{5}$ to $\frac{2}{3}$.

This restriction is justified by the results of the analysis of the single digit probabilities of Adder 2 by Rohatsch (21). He discovered that the probability of zero digits in the sum representation varied little from $\frac{1}{2}$ when the probability of zero digits of both operand representations varied from 0 to 1.

This analysis takes a somewhat different tack than that taken by Rohatsch. He analyzed the output representations on the basis of

the triplet* probabilities of both operand representations, where the representations of both operands were defined parametrically. In the present analysis, only one operand, $\emptyset$, is parametrically defined. The other operand, A, is assumed to have the same distribution as the sum representation, A'. This is justified by the observation that at least one of the operands taking part in an addition is the sum of a previous addition in practically all of the uses of the adder in a typical calculation. The transfer digits are assumed to reach a steady-state distribution independent of digital position.

Each possible combination of digits for the A operand and the input transfers (t_{i+1}, t'_{i+1}) was identified as the present state. Table 11 lists these, indicating which states are equivalent because of the symmetry of the adder. Each possible combination of digits for the sum and the output transfers (t_{i-1}, t'_{i-1}) was identified as the next state, also employing Table 11, with the following equivalences

$$t_{i-1} \cong t_{i+1} \qquad \text{(A1.1)}$$

$$t'_{i-1} \cong t'_{i+1} \qquad \text{(A1.2)}$$

$$a'_j \cong a_j \qquad j=i,\ i+1 \qquad \text{(A1.3)}$$

---

*Three consecutive digits of the number.

Table 11. States of the Markov Process Used to Analyze Adder 3.

State	'Positive' State				'Negative' State				Steady State No.
	t_{i+1}	t'_{i+1}	a_i	a_{i+1}	t_{i+1}	t'_{i+1}	a_i	a_{i+1}	
A	1	0	1	1	$\bar{1}$	0	$\bar{1}$	$\bar{1}$	-
B	1	0	1	0	$\bar{1}$	0	$\bar{1}$	0	-
C	1	0	1	$\bar{1}$	$\bar{1}$	0	$\bar{1}$	1	-
D	1	0	0	1	$\bar{1}$	0	0	$\bar{1}$	-
E	1	0	0	0	$\bar{1}$	0	0	0	1
F	1	0	0	$\bar{1}$	$\bar{1}$	0	0	1	2
G	1	0	$\bar{1}$	1	$\bar{1}$	0	1	$\bar{1}$	3
H	1	0	$\bar{1}$	0	$\bar{1}$	0	1	0	4
I	1	0	$\bar{1}$	$\bar{1}$	$\bar{1}$	0	1	1	5
J	1	$\bar{1}$	1	1	$\bar{1}$	1	$\bar{1}$	$\bar{1}$	-
K	1	$\bar{1}$	1	0	$\bar{1}$	1	$\bar{1}$	0	6
L	1	$\bar{1}$	1	$\bar{1}$	$\bar{1}$	1	$\bar{1}$	1	7
M	1	$\bar{1}$	0	1	$\bar{1}$	1	0	$\bar{1}$	8*A*
N	1	$\bar{1}$	0	0	$\bar{1}$	1	0	0	9
O	1	$\bar{1}$	0	$\bar{1}$	$\bar{1}$	1	0	1	10
P	1	$\bar{1}$	$\bar{1}$	1	$\bar{1}$	1	1	$\bar{1}$	-
Q	1	$\bar{1}$	$\bar{1}$	0	$\bar{1}$	1	1	0	-
R	1	$\bar{1}$	$\bar{1}$	$\bar{1}$	$\bar{1}$	1	1	1	-
S	0	0	1	1	0	0	$\bar{1}$	$\bar{1}$	-
T	0	0	1	0	0	0	$\bar{1}$	0	11
U	0	0	1	$\bar{1}$	0	0	$\bar{1}$	1	12
V	0	0	0	1	0	0	0	$\bar{1}$	8*B*
W	0	0	0	0	-	-	-	-	13

Table 12. Transition Probabilities of the Persistant States.

Next State	Present State												
	1	2	3	4	5	6	7	8	9	10	11	12	13
1	0	0	0	0	$P1$	$P1$	0	0	0	0	$P1$	0	0
2	0	0	0	$P1$	$P3$	$P3$	$P1$	$P1$	0	0	$P1$	0	0
3	$P1$	0	0	0	0	0	0	0	0	$P1$	$P4$	$P1$	0
4	0	$P1$	$P1$	$P3$	$P2$	$P2$	$P3$	$P4$	$P2$	$P4$	$P2$	$P3$	$P2$
5	$P1$	$P4$	$P3$	$P2$	$P1$	0	0	0	0	0	0	0	0
6	$P4$	$P1$	0	0	0	0	0	0	0	0	0	0	0
7	0	0	0	0	0	$P1$	$P2$	0	$P4$	$P1$	0	0	0
8	$P2$	$P3$	$P4$	$P1$	0	0	$P1$	$P1$	$P3$	$P2$	$P1$	$P2$	$2\times P4$
9	$P4$	0	$P2$	$P1$	0	0	$P1$	$P4$	0	$P4$	0	$P1$	0
10	0	0	$P1$	0	0	0	0	0	0	0	0	0	0
11	$P1$	0	$P1$	0	0	0	0	$P1$	0	$P1$	0	0	0
12	0	$P1$	0	0	$P4$	$P4$	0	$P2$	$P1$	0	$P4$	$P1$	$P2$
13	$P1$	$P2$	0	$P4$	$P1$	$P1$	$P4$	$P1$	$P2$	$P1$	$P1$	$P4$	$P2$

The transition probabilities were then determined, based on the probabilities of pairs of the $\emptyset$ operand. The determination of the steady-state digit statistics can then be determined the steady-state distribution of states in the Markov process defined above. Examination of the transition matrix showed that fourteen of the twenty-three states are persistent, and that states M and V are equivalent with respect to their transition probabilities to subsequent states. This is also indicated in Table 11. The transition probabilities of the persistent states are given as Table 12. In this table,

$$P1 = \frac{z}{4} \tag{A1.4}$$

$$P2 = \frac{z}{2} \tag{A1.5}$$

$$P3 = \frac{1}{2} - \frac{z}{2} \tag{A1.6}$$

$$P4 = \frac{1}{2} - \frac{3z}{4} \tag{A1.7}$$

where

z is the probability of zero of the parametrically defined input representation, $\frac{2}{5} \leq z < \frac{2}{3}$.

The probabilities of the two equivalent states can be determined by finding the probability of state V (or 8b) and subtracting that from

Table 13. Pair Probabilities of Sums Produced by Adder 3.

z	(0,0)	$(0,1)$ $(0,\bar{1})$	$(1,\bar{1})$ $(\bar{1},1)$	$(1,0)$ $(\bar{1},0)$	$(1,1)$ $(\bar{1},\bar{1})$
.40	.256339	.129070	.078276	.128170	.036316
.41	.255566	.128686	.079022	.127783	.036726
.42	.254825	.128297	.079753	.127413	.037125
.43	.254117	.127902	.080468	.127059	.037513
.44	.253441	.127501	.081167	.126721	.037891
.45	.252795	.127095	.081851	.126398	.038258
.46	.252180	.126684	.082520	.126090	.038616
.47	.251593	.126269	.083173	.125796	.038965
.48	.251034	.125850	.083810	.125517	.039306
.49	.250504	.125427	.084431	.125252	.039638
.50	.250000	.125000	.085037	.125000	.039963
.51	.249523	.124570	.085626	.124761	.040281
.52	.249071	.124136	.086200	.124535	.040593
.53	.248644	.123700	.086757	.124322	.040899
.54	.248243	.123261	.087298	.124121	.041199
.55	.247865	.122819	.087822	.123932	.041493
.56	.247511	.122375	.088330	.123756	.041784
.57	.247180	.121929	.088821	.123590	.042069
.58	.246872	.121480	.089296	.123436	.042351
.59	.246587	.121030	.089753	.123294	.042630
.60	.246324	.120578	.090193	.123162	.042906
.61	.246082	.120124	.090615	.123041	.043179
.62	.245863	.119668	.091020	.122931	.043450
.63	.245664	.119210	.091407	.122832	.043719
.64	.245486	.118751	.091775	.122743	.043988
.65	.245330	.118290	.092125	.122665	.044255
.66	.245194	.117827	.092457	.122597	.044523

the probability of state 8 to find the probability of state M (or 8a). The equations to accomplish this are

$$D(8b) = P2 \cdot D(1)+D(10) + P1 \cdot D(2)+D(4)+D(7)+D(9) + P4 \cdot D(3) \quad (A1.8)$$

$$D(8a) = D(8) - D(8b) \quad (A1.9)$$

where

D(x) is the probability of state x.

Tables 13, 14, and 15 present the results of the analysis. Table 13 gives the pair probabilities, where

$$(x,y) = P(a'_i = x,\ a'_{i+1} = y).$$

Note that at z = 0.5,

$$P(a'_i = 0) = (0,0) + (0,1) + (0,\bar{1}) = 0.500000 \quad (A1.10)$$

and

$$P(a'_{i+1} = 0) = (0,0) + (1,0) + (\bar{1},0) = 0.500000 \quad (A1.11)$$

Since each of these is the probability of zero digits in sum representations, the steady-state probabilities of the sum as given by $z = \frac{1}{2}$ closely approximates the steady-state probability when both operands are the result of previous additions. Calculations similar to (A1.10) and (A1.11) for z = 0.40 yield that

$$P(a'_i = 0) = 0.514479,\ P(a'_{i+1} = 0) = 0.512679 \quad (A1.12)$$

Table 14. Conditional Probabilities of Digits in the Sums Produced by Adder 3.

z	$(0\,\vert\,0)$	$(1\,\vert\,0)$ $(\bar{1}\,\vert\,0)$	$(\bar{1}\,\vert\,1)$ $(1\,\vert\,\bar{1})$	$(0\,\vert\,1)$ $(0\,\vert\,\bar{1})$	$(1\,\vert\,1)$ $(\bar{1}\,\vert\,\bar{1})$
.40	.498251	.250875	.322440	.527967	.149594
.41	.498238	.250881	.324484	.524709	.150808
.42	.498271	.250865	.326466	.521562	.151972
.43	.498347	.250827	.328386	.518523	.153091
.44	.498465	.250767	.330246	.515588	.154166
.45	.498624	.250688	.332045	.512754	.155201
.46	.498824	.250588	.333783	.510019	.156198
.47	.499062	.250469	.335463	.507378	.157159
.48	.499338	.250331	.337083	.504830	.158087
.49	.499651	.250175	.338644	.502371	.158984
.50	.500000	.250000	.340147	.500000	.159853
.51	.500384	.249808	.341591	.497713	.160696
.52	.500803	.249599	.342976	.495509	.161514
.53	.501254	.249373	.344304	.493386	.162311
.54	.501739	.249130	.345573	.491340	.163087
.55	.502256	.248872	.346783	.489371	.163845
.56	.502804	.248598	.347936	.487477	.164587
.57	.503383	.248308	.349030	.485656	.165315
.58	.503993	.248004	.350065	.483906	.166030
.59	.504632	.247684	.351041	.482225	.166734
.60	.505301	.247349	.351957	.480612	.167430
.61	.505999	.247000	.352814	.479067	.168119
.62	.506726	.246637	.353611	.477587	.168803
.63	.507482	.246259	.354347	.476171	.169483
.64	.508266	.245867	.355021	.474818	.170161
.65	.509079	.245460	.355634	.473527	.170840
.66	.509921	.245040	.356183	.472296	.171520

Table 15. Transfer Digit Probabilities in Adder 3.

z	(0,0)	$(1,\bar{1})$ $(\bar{1},1)$	$(1,0)$ $(\bar{1},0)$
.40	.351448	.126288	.197988
.41	.353988	.124556	.198450
.42	.356484	.122849	.198909
.43	.358938	.121167	.199363
.44	.361351	.119510	.199815
.45	.363722	.117875	.200264
.46	.366054	.116262	.200711
.47	.368347	.114671	.201156
.48	.370602	.113099	.201600
.49	.372819	.111547	.202043
.50	.375000	.110014	.202486
.51	.377145	.108498	.202930
.52	.379256	.106998	.203374
.53	.381333	.105514	.203819
.54	.383377	.104045	.204266
.55	.385389	.102590	.204716
.56	.387369	.101148	.205168
.57	.389318	.099718	.205623
.58	.391238	.098299	.206081
.59	.393129	.096891	.206544
.60	.394992	.095493	.207011
.61	.396827	.094103	.207484
.62	.398635	.092721	.207962
.63	.400417	.091345	.208446
.64	.402174	.089976	.208937
.65	.403907	.088611	.209435
.66	.405615	.087251	.209941

Calculations for $z = 0.60$ indicate that

$$P(a'_i = 0) = 0.487480, \quad P(a'_{i+1} = 0) = 0.492648 \qquad \text{(A1.13)}$$

The small deviation from $\frac{1}{2}$ illustrates the strong tendency that the sum representation of this adder have to $P(a'_i = 0) = \frac{1}{2}$.

Table 14, gives the conditional probabilities for the steady-state sum representation, where

$$P(x \;\; y) = P(a'_{i+1} = x \;\; a'_i = y) \qquad \text{(A1.14)}$$

The entries for $z = \frac{1}{2}$, which are required to evaluate the various normalization strategies, can be approximated by very simple fractions. The approximations are $\frac{1}{2}, \frac{1}{4}, \frac{1}{3}, \frac{1}{2}, \frac{1}{6}$ in the order of their appearance. These values are employed in the calculations to evaluate the possible normalization strategies.

Table 15 gives the probability of occurrence of the various transfer digit combinations, where

$$(x, y) = P(t_i = x, \; t'_i = y). \qquad \text{(A1.15)}$$

APPENDIX II

MINIMAL RIGHT-DIRECTED RECODER OF RADIX TWO SIGNED DIGIT NUMBERS

Table 16 gives the overall right-directed recoder which is the result of applying the extended Penhollow recoder (Table 6) to the output of the assimilator of Table 5. This recoder should be symmetric*, since the numbers to be recoded are symmetric. Table 16 was checked for symmetry and ten lines** did not have symmetric mates. It was found that altering any of the ten lines to make it part of a symmetric pair did not alter the probability of zero digits in the recoder output. Hence, the recoded digit and mode digit of these ten lines are pseudo don't-care conditions. They can be chosen from two of the three possible values and still not affect the probability of zero digits (e.g., the shift average) of the recoder output. Viewed in this way, Table 16 is one member of a class of minimal recoders that contains 2^{10} members.

---

*That is, for each line in the table a second line should be found whose entries are the negative of the entries of the first line.

**Lines 5,27; 12,34; 17,39; 49,65; 54,70. The pairs separated by semi-colons are the pairs which should be symmetric.

Table 16. Recoder Based on the Cascaded Assimilator-Recoder of Tables 5 and 6.

Line No.	Choose		Known					
	m_i	z_i	m_{i-1}	x_i	x_{i+1}	x_{i+2}	x_{i+3}	x_{i+4}
0	0	0	0	0	0	-	-	-
1	0	0	0	0	1	$\bar{1}$	-	-
2	0	0	0	0	1	0	-	-
3	0	0	0	0	1	1	$\bar{1}$	-
4	0	0	0	0	1	1	0	$\bar{1}$
5	0	0	0	0	1	1	0	0
6	1	1	0	0	1	1	0	1
7	1	1	0	0	1	1	1	-
8	$\bar{1}$	0	0	1	$\bar{1}$	$\bar{1}$	-	-
9	$\bar{1}$	0	0	1	$\bar{1}$	0	-	-
10	$\bar{1}$	0	0	1	$\bar{1}$	1	$\bar{1}$	-
11	$\bar{1}$	0	0	1	$\bar{1}$	1	0	$\bar{1}$
12	$\bar{1}$	0	0	1	$\bar{1}$	1	0	0
13	0	1	0	1	$\bar{1}$	1	0	1
14	0	1	0	1	$\bar{1}$	1	1	-
15	$\bar{1}$	0	0	1	0	$\bar{1}$	$\bar{1}$	-
16	$\bar{1}$	0	0	1	0	$\bar{1}$	0	$\bar{1}$
17	$\bar{1}$	0	0	1	0	$\bar{1}$	0	0
18	0	1	0	1	0	$\bar{1}$	0	1
19	0	1	0	1	0	$\bar{1}$	1	-
20	0	1	0	1	0	0	-	-
21	0	1	0	1	0	1	-	-
22	0	1	0	1	1	-	-	-
23	0	0	0	0	$\bar{1}$	1	-	-
24	0	0	0	0	$\bar{1}$	0	-	-
25	0	0	0	0	$\bar{1}$	$\bar{1}$	1	-
26	0	0	0	0	$\bar{1}$	$\bar{1}$	0	1
27	$\bar{1}$	$\bar{1}$	0	0	$\bar{1}$	$\bar{1}$	0	0
28	$\bar{1}$	$\bar{1}$	0	0	$\bar{1}$	$\bar{1}$	0	$\bar{1}$
29	$\bar{1}$	$\bar{1}$	0	0	$\bar{1}$	$\bar{1}$	$\bar{1}$	-
30	1	0	0	$\bar{1}$	1	1	-	-
31	1	0	0	$\bar{1}$	1	0	-	-
32	1	0	0	$\bar{1}$	1	$\bar{1}$	1	-
33	1	0	0	$\bar{1}$	1	$\bar{1}$	0	1
34	0	$\bar{1}$	0	$\bar{1}$	1	$\bar{1}$	0	0
35	0	$\bar{1}$	0	$\bar{1}$	1	$\bar{1}$	0	$\bar{1}$
36	0	$\bar{1}$	0	$\bar{1}$	1	$\bar{1}$	$\bar{1}$	-
37	1	0	0	$\bar{1}$	0	1	1	-
38	1	0	0	$\bar{1}$	0	1	0	1

Table 16 (Continued).

Line No.	Choose		Known					
	m_i	z_i	m_{i-1}	x_i	x_{i+1}	x_{i+2}	x_{i+3}	x_{i+4}
39	0	$\bar{1}$	0	$\bar{1}$	0	1	0	0
40	0	$\bar{1}$	0	$\bar{1}$	0	1	0	$\bar{1}$
41	0	$\bar{1}$	0	$\bar{1}$	0	1	$\bar{1}$	-
42	0	$\bar{1}$	0	$\bar{1}$	0	0	-	-
43	0	$\bar{1}$	0	$\bar{1}$	0	$\bar{1}$	-	-
44	0	$\bar{1}$	0	$\bar{1}$	$\bar{1}$	-	-	-
45	1	0	1	1	1	1	-	-
46	1	0	1	1	1	0	-	-
47	1	0	1	1	1	$\bar{1}$	1	-
48	1	0	1	1	1	$\bar{1}$	0	1
49	0	$\bar{1}$	1	1	1	$\bar{1}$	0	0
50	0	$\bar{1}$	1	1	1	$\bar{1}$	0	$\bar{1}$
51	0	$\bar{1}$	1	1	1	$\bar{1}$	$\bar{1}$	-
52	1	0	1	1	0	1	1	-
53	1	0	1	1	0	1	0	1
54	0	$\bar{1}$	1	1	0	1	0	0
55	0	$\bar{1}$	1	1	0	1	0	$\bar{1}$
56	0	$\bar{1}$	1	1	0	1	$\bar{1}$	-
57	0	$\bar{1}$	1	1	0	0	-	-
58	0	$\bar{1}$	1	1	0	$\bar{1}$	-	-
59	0	$\bar{1}$	1	1	$\bar{1}$	-	-	-
60	1	$\bar{1}$	1	0	1	-	-	-
61	$\bar{1}$	0	$\bar{1}$	$\bar{1}$	$\bar{1}$	$\bar{1}$	-	-
62	$\bar{1}$	0	$\bar{1}$	$\bar{1}$	$\bar{1}$	0	-	-
63	$\bar{1}$	0	$\bar{1}$	$\bar{1}$	$\bar{1}$	1	$\bar{1}$	-
64	$\bar{1}$	0	$\bar{1}$	$\bar{1}$	$\bar{1}$	1	0	$\bar{1}$
65	$\bar{1}$	0	$\bar{1}$	$\bar{1}$	$\bar{1}$	1	0	0
66	0	1	$\bar{1}$	$\bar{1}$	$\bar{1}$	1	0	1
67	0	1	$\bar{1}$	$\bar{1}$	$\bar{1}$	1	1	-
68	$\bar{1}$	0	$\bar{1}$	$\bar{1}$	0	$\bar{1}$	$\bar{1}$	-
69	$\bar{1}$	0	$\bar{1}$	$\bar{1}$	0	$\bar{1}$	0	$\bar{1}$
70	$\bar{1}$	0	$\bar{1}$	$\bar{1}$	0	$\bar{1}$	0	0
71	0	1	$\bar{1}$	$\bar{1}$	0	$\bar{1}$	0	1
72	0	1	$\bar{1}$	$\bar{1}$	0	$\bar{1}$	1	-
73	0	1	$\bar{1}$	$\bar{1}$	0	0	-	-
74	0	1	$\bar{1}$	$\bar{1}$	0	1	-	-
75	0	1	$\bar{1}$	$\bar{1}$	1	-	-	-
76	$\bar{1}$	1	$\bar{1}$	0	$\bar{1}$	-	-	-

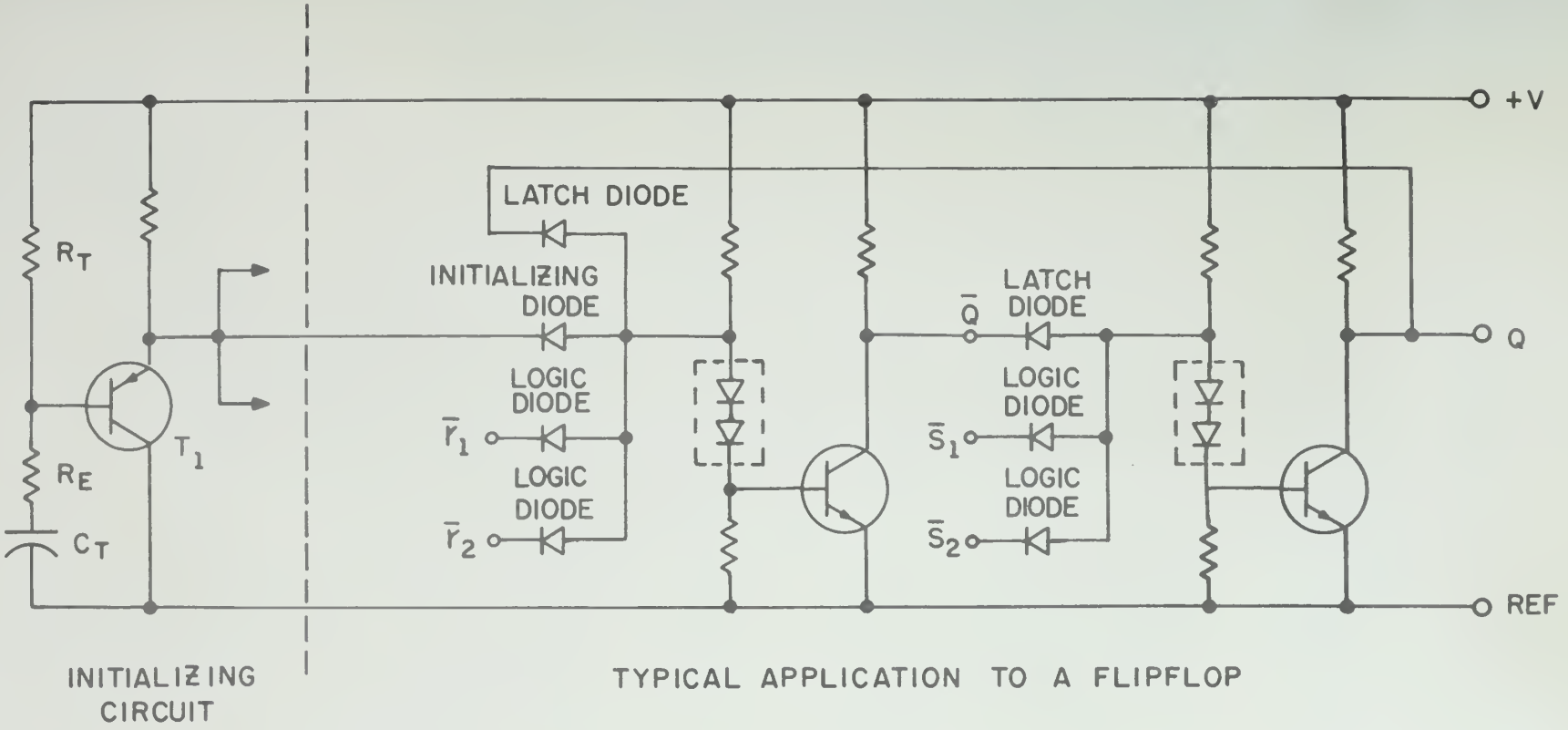


Figure 20. Self-initializing circuit.

APPENDIX III

SELF-INITIALIZING MODULES

Initializing a limited connection arithmetic unit with a centralized circuit requires that each module have one electrical connection for the initializing signal. Since one of the major requirements of LSI is minimizing the number of connections to the module, these pins should be eliminated if possible.

In one method which does not require pins for initializing signals, a circuit which activates an initializing signal for a fixed period of time after power is applied is placed on each module. This signal is distributed to all points on the module that require an initializing signal. An example of this technique is shown in Figure 20, where the initializing signal is active while C_T is charging.

REFERENCES

1. Allard, R. W., Wolf, K. A., and Zemlin, R. A., "Some Effects of the 6600 Computer on Language Structures," Communications of the ACM, Vol. 7 No. 2, Feb. 1964, pp. 112-119.

2. Anderson, S. F., et al., "The IBM System/360 Model 91: Floating Point Execution Unit," IBM Journal of Research and Development, Vol. 11 No. 1, Jan. 1967, pp 34-53.

3. Atkins, D. C. III, "The Theory and Implementation of SRT Division," Report 230, Department of Computer Science, University of Illinois, Urbana, Illinois, June 1967.

4. Avizienis, Algirdas, "A Study of Redundant Number Representations for Parallel Digital Computers," Report 101, Department of Computer Science, University of Illinois, Urbana, Illinois, May 1960.

5. _____, "Signed Digit Number Representation for Fast Parallel Arithmetic," IRE Transactions on Electronic Computers, Vol. EC-10 No. 3, Sept. 1961, pp. 389-400.

6. _____, "A Flexible Implementation of Digital Computer Arithmetic," Information Processing 1962, North Holland Publishing Company, Amsterdam, 1962, pp. 664-670.

7. Borovec, R. T., "The Logical Design of a Class of Limited Carry-Borrow Propogation Adders," Report 275, Department of Computer Science, University of Illinois, Urbana, Illinois, August 1968.

8. Comfort, W. T., Private Communication, July 1968.

9. Elspas, B., et al., "Investigation of Propogation-Limited Computer Networks," AFCRL-64-376, Stanford Research Institute, Menlo Park, California, April 1964.

10. _____, "Investigation of Propogation-Limited Computer Networks," AFCRL-64-376 (II), Stanford Research Institute, Menlo Park, California, July 1965.

11. IBM Corporation, "IBM 7094 Principles of Operation", A22-6703, Poughkeepsie, New York, 1962.

12. Muller, D. E., "Asynchronous Logics and Application to Information Processing," Switching Theory in Space Technology, Ed.: Aiken and Main, Stanford University Press, 1963, pp. 289-297.

13. Nakata, I., "A Note on Compiling Algorithms for Arithmetic Expressions," Communications of the ACM, Vol. 10 No. 8, Aug. 1967, pp. 492-494.

14. Nash, J. P., Ed., "Illiac Programming, A Guide to the Preparation of Problems for Solution by the University of Illinois Digital Computer," Department of Computer Science, University of Illinois, Urbana, Illinois, 1956.

15. Penhollow, J. O., "A Study of Arithmetic Recodings with Applications in Multiplication and Division," Report 128, Department of Computer Science, University of Illinois, Urbana, Illinois, Sept. 1962.

16. Robertson, J. E., "A Deterministic Procedure for the Design of Carry-Save Adders and Borrow-Save Subtractors," Report 235, Department of Computer Science, University of Illinois, Urbana, Illinois, July 1967.

17. _____, "A New Class of Digital Division Methods," IRE Transactions on Electronic Computers, Vol. EC-7 No. 3, Sept. 1958, pp. 218-222.

18. _____, "Methods of Selecting Quotient Digits During Division," Proceedings of IFIP Congress 1965, Spartan Books, Washington, D. C., 1966, pp. 444-445.

19. _____, "The Correspondence Between Methods of Digital Division and Multiplier Recoding Procedures," Report 252, Department of Computer Science, University of Illinois, Urbana, Illinois, Dec. 1967.

20. _____, Private Communication, July 1969.

21. Rohatsch, F. A. , "A Study of Transformations Applicable to the Development of Limited Carry-Borrow Propogation Adders," Report 226, Department of Computer Science, University of Illinois, Urbana, Illinois, June 1967.

22. Stone, H. S., "One-Pass Compilation of Arithmetic Expressions for a Parallel Processor," Communications of the ACM, Vol. 10 No. 4, April 1967, pp. 220-223.

23. Thorlin, J. F., "Code Generation for PIE (Parallel Instruction Execution) Computer," AFIPS Conference Proceedings, 1967 Spring Joint Computer Conference, Thompson Book Company, Washington, D.C., 1967, pp. 641-643.

24. Thomasulo, R. M., "An Efficient Algorithm for Exploiting Multiple Arithmetic Units," IBM Journal of Research and Development, Vol. 11 No. 1, Jan. 1967, pp. 25-33.

VITA

Michael John Pisterzi was born in Chicago, Illinois, on September 28, 1936.

He was awarded the Bachelor of Science degree, with High Honors, in Electrical Engineering by the University of Illinois in June 1961. He received the Honeywell Award and the University of Illinois Outstanding Electrical Engineering Student Award the year of his graduation. He received the Master of Science degree in Electrical Engineering from the University of Illinois in June 1962.

He is on the staff of the System Development Division of the IBM Corporation. He has had positions with the Admiral Corporation, Motorola, Incorporated, and the Douglas Aircraft Company. He has had teaching appointments with the Electrical Engineering Department of the University of Illinois and research appointments with the Department of Computer Science of the University of Illinois.

He is a member of Phi Kappa Phi, Tau Beta Pi, Signma Tau, Eta Kappa Nu, Pi Mu Epsilon, Chi Gamma Iota, Phi Alpha Mu, the Institute of Electrical and Electronic Engineers, and the Association for Computing Machinery.